MySQL High Availability



高可用MySQL

构建健壮的数据中心

Charles Bell, Mats Kindahl
& Lars Thalmann 著
Mark Callaghan 作序
宁青 唐李洋 诸云萍 译

O'REILLY®
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

高可用MySQL——构建健壮的数据中心

服务器瓶颈和故障是数据库部署中的常见问题，但并不是没有解决的办法。

MySQL的一些特性能够保证，无论系统运行在硬件、虚拟机，还是云环境，系统都不停机。《高可用MySQL》一书讲解了真实环境下如何使用这些复制、集群和监控特性。

本书由设计这些工具的工程师们所编写，揭示了MySQL可靠性和高可用性一些难以发现的方方面面，这些知识对使用该数据库系统的组织来说非常重要。

- 探究二进制日志，这是在灾难恢复和故障诊断的时候需要复制的文件；
- 掌握改善响应时间和处理大数据集的技术；
- 学习如何使用复制横向扩展服务器；
- 监控数据库活动和性能，以及主要的操作系统参数；
- 跟踪master和slave，处理故障、重启、崩溃及其他事故；
- 使用作者写的开源库实现关键任务的自动化；
- 学习虚拟化环境下的MySQL技术，如Amazon Web Services；
- 使用MySQL集群实现高可用性。

建议阅读本书的读者具有一定的编程经验。

“MySQL复制被广泛部署，但却从未得到充分的解释。这本书将改变这种状况。”

——Mark Callaghan
MySQL贡献者，
同时是一些世界级
Internet公司的
MySQL工程领导

Charles Bell博士是复制和备份领域的高级开发人员，兴趣包括数据库理论，软件工程及敏捷开发实践。

Mats Kindahl博士是MySQL复制和备份小组的首席开发者。

Lars Thalmann博士是MySQL复制和备份小组的开发经理和技术领导，设计了很多复制和备份特性。

O'REILLY®
oreilly.com.cn

图书分类：数据库>MySQL

策划编辑：张春雨
责任编辑：高洪霞

Broadview®
WWW.BROADVIEW.COM.CN
www.phei.com.cn

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-14407-3
9 787121 144073 >

定价：98.00元

O'REILLY®

高可用MySQL

——构建健壮的数据中心

MySQL High Availability

Charles Bell, Mats Kindahl, Lars Thalmann 著
Mark Callaghan 作序
宁青 唐李洋 诸云萍 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是“MySQL High Availability”的中文翻译版，主要讲解真实环境下如何使用 MySQL 的复制、集群和监控特性，揭示 MySQL 可靠性和高可用性的方方面面。本书由 MySQL 开发团队亲自执笔，定位于解决 MySQL 数据库的常见应用瓶颈，在保持 MySQL 的持续可用性的前提下，挖潜各种提高性能的解决方案。本书分为三个部分。第一部分讲述 MySQL 复制，包括高可用性和横向扩展，第二部分介绍构建健壮的数据中心时监控和性能方面的问题，第三部分给出其他 MySQL 相关内容，包括云计算和 MySQL 集群。

本书读者对象是 MySQL 专业人士。假设读者已拥有 SQL、MySQL 管理和操作系统的基础背景知识。书中介绍一些关于复制、灾难恢复、系统监控及其他高可用性主题的背景信息。相关有用的背景知识请参考其他书籍的第 1 章。对于相关专业的师生，本书也有很高的参考价值。

978-0-596-80730-6 MySQL High Availability © 2010 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2010.Authorized translation of the English edition, 2010 O'Reilly Media, Inc., the owner of all rights to publish and sell the same. All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2011-5406

图书在版编目（CIP）数据

高可用 MySQL：构建健壮的数据中心 /（美）贝尔（Bell,C.），（美）肯德尔（Kindahl,M.），（美）塞尔曼（Thalmann,L.）著；宁青，唐李洋，诸云萍译. —北京：电子工业出版社，2011.10
书名原文：MySQL High Availability
ISBN 978-7-121-14407-3

Ⅰ. ①高…　Ⅱ. ①贝…　②肯…　③塞…　④宁…　⑤唐…　⑥诸…　Ⅲ. ①关系数据库—数据库管理系统，MySQL　Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2011）第 168169 号

策划编辑：张春雨
责任编辑：高洪霞
封面设计：Karen Montgomery　张　健
印　　刷：
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：37.75　字数：785 千字
印　　次：2011 年 10 月第 1 次印刷
印　　数：4000 册　　定价：98.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

Table of Contents

前言 i

第一部分 复制

第1章 引言 3

到底什么是复制 5

那么，是否需要备份 6

什么是监控 7

还有什么我可以阅读的 7

小结 8

第2章 MySQL复制原理 9

复制的基本步骤 10

配置Master 11

配置Slave 12

连接Master和Slave 13

二进制日志简介 14

二进制日志记录了什么 15

观察复制的动作 16

二进制日志的结构和内容 17

使用Python管理复制 20

基本类及函数……22
操作系统……23
服务器类……23
服务器角色……25
建立新Slave……26
克隆Master……27
克隆Slave……29
克隆操作的脚本……31
执行常见的复制任务……33
报表……33
小结……39
第3章 二进制日志……41
二进制日志的结构……42
Binlog事件的结构……44
记录语句……45
记录数据操作语言……46
记录数据定义语言……46
记录查询……47
LOAD DATA INFILE语句……52
二进制日志过滤器……54
触发器、事件和存储程序……56
存储过程……61
存储函数……64
Events……66
特殊结构……66
非事务性的变化和错误处理……67
记录事务……70
事务缓存……71
使用XA进行分布式事务处理……74

二进制日志管理……76
二进制日志和系统崩溃安全……77
Binlog文件轮换（Rotation）……78
事故（incident）……79
清除binlog文件……80
mysqlbinlog实用工具……81
基本用法……81
解释事件（Interpreting Events）……88
二进制日志选项和变量……92
小结……94
第4章 基于复制的数据库高可用技术……95
冗余……96
计划……98
Slave故障……98
Master故障……98
中继服务器故障……99
灾难恢复……99
程序……99
热备份……102
双Master……107
半同步复制……116
Slave的提升……119
循环复制……134
小结……138
第5章 MySQL集群的横向扩展……139
读操作的横向扩展……141
异步复制的价值……142
管理复制拓扑……144

应用层负载均衡器示例 …… 147
级联复制（Hierarchal Replication） …… 150
中继服务器的设置 …… 151
使用Python添加中继服务器 …… 152
专用Slave …… 153
过滤复制事件 …… 154
使用过滤将事件分配给Slave …… 155
数据分片 …… 157
分片的表示 …… 159
数据分区 …… 160
分片之间的均衡 …… 161
一个分片的例子 …… 163
数据的一致性管理 …… 174
非级联部署中的一致性 …… 174
级联部署中的一致性 …… 176
小结 …… 182

第6章 高级复制 …… 183

复制架构基础 …… 184
中继日志的结构 …… 185
复制线程 …… 187
Slave线程的启动和停止 …… 188
通过Internet运行复制 …… 189
使用内置支持建立安全复制 …… 191
使用Stunnel建立安全复制 …… 192
细粒度控制复制 …… 194
关于复制状态的信息 …… 194
处理断开连接的参数 …… 201
Slave是如何处理事件的 …… 202
管理I/O线程 …… 202

SQL线程的处理 ····· 203
Slave的安全和恢复 ····· 208
同步，事务和数据库崩溃问题 ····· 209
保护非事务性语句的规则 ····· 210
多源复制 ····· 211
基于行的复制 ····· 214
基于行的复制参数 ····· 215
混合模式的复制 ····· 215
处理基于行复制的事件 ····· 216
事件的执行 ····· 220
事件和触发器 ····· 222
过滤 ····· 223
小结 ····· 225

第二部分 监控和灾难恢复

第7章 监控入门 ····· 229
监控方法 ····· 230
监控的好处 ····· 231
监控系统组件 ····· 231
处理器 ····· 231
内存 ····· 233
磁盘 ····· 233
网络子系统 ····· 234
监控方法 ····· 235
Linux和UNIX监控 ····· 236
进程活动 ····· 237
内存利用率 ····· 241
磁盘利用率 ····· 243
网络活动 ····· 246

常见系统统计信息 ····· 248
使用cron自动监控 ····· 249
Mac OS X 监控 ····· 249
System profile ····· 250
控制台（Console） ····· 252
Activity Monitor ····· 253
Microsoft Windows监控 ····· 257
Windows Experience ····· 257
System Health Report ····· 259
Event Viewer（事件查看器） ····· 261
Reliability Monitor ····· 263
The Task Manager（任务管理器） ····· 264
Performance Monitor ····· 266
预防性维护监控 ····· 267
小结 ····· 268

第8章 监控MySQL ····· 269

什么是性能 ····· 270
MySQL服务器监控 ····· 270
如何显示MySQL性能 ····· 271
性能监控 ····· 271
SQL命令 ····· 272
mysqladmin实用工具 ····· 278
MySQL GUI工具 ····· 279
MySQL管理器 ····· 280
MySQL查询浏览器 ····· 289
服务器日志 ····· 290
第三方工具 ····· 293
MySQL Benchmark套件 ····· 295

数据库性能 ····· 296
测量数据库性能 ····· 297
数据库优化的最佳实践 ····· 308
提高性能的最佳实践 ····· 315
一切都慢 ····· 316
慢查询 ····· 316
慢应用 ····· 316
慢复制 ····· 317
小结 ····· 317

第9章 监控存储引擎 ····· 319

MyISAM ····· 320
优化磁盘存储 ····· 320
优化数据库表 ····· 320
使用MyISAM实用工具 ····· 321
按索引顺序存储表 ····· 323
压缩表 ····· 323
对数据表进行碎片整理 ····· 323
监控Key Cache ····· 324
预加载Key Cache ····· 325
使用多个Key Cache ····· 325
其他需要考虑的参数 ····· 326
InnoDB ····· 328
使用SHOW ENGINE命令 ····· 329
使用InnoDB监控器 ····· 332
监控日志文件 ····· 335
监控缓冲池 ····· 336
监控表空间 ····· 338
使用INFORMATION_SCHEMA表 ····· 338

其他需要考虑的参数……339
小结……340
第10章 复制监控……341
开始……341
安装服务器……342
包容性和排他性复制……342
复制线程……344
监控Master……346
Master的监控命令……346
Master的状态变量……349
监控Slave……350
Slave的监控命令……350
Slave的状态变量……353
使用MySQL管理器监控复制……354
其他需要考虑的项……356
网络……356
监控和管理Slave滞后……356
Slave延迟的原因和预防措施……357
小结……358
第11章 复制的故障排除……361
什么导致错误发生……362
Master上的问题……362
Slave上的问题……366
高级复制问题……371
排除复制故障的工具……372
最佳实践……374
了解你的拓扑结构……374
查询所有服务器的状态……376

查看日志 ····· 376
检查配置信息 ····· 377
有序执行系统关闭 ····· 377
在遇到错误后按序执行重启 ····· 377
手动执行失败查询 ····· 378
常用程序 ····· 378
报告复制故障 ····· 379
小结 ····· 380

第12章 保护你的投资 ····· 383

什么是信息保障 ····· 384
信息保障的三个应用 ····· 384
信息保障为什么重要 ····· 385
信息完整性、灾难恢复及备份的职责 ····· 385
高可用性与灾难恢复 ····· 386
灾难恢复 ····· 386
数据恢复的重要性 ····· 391
备份和恢复 ····· 392
备份实用程序和操作系统级的解决方案 ····· 396
InnoDB Hot Backup应用 ····· 397
物理文件的复制 ····· 400
mysqldump工具 ····· 402
XtraBackup ····· 404
逻辑卷管理器快照 ····· 404
备份方法的比较 ····· 409
备份和MySQL复制 ····· 410
使用复制进行备份和恢复 ····· 410
PITR ····· 411
自动备份 ····· 418

小结 ····· 421

第13章 MySQL企业版 ····· 423

MySQL企业版入门 ····· 424

订阅级别 ····· 425

安装概述 ····· 426

MySQL企业组件 ····· 427

MySQL企业服务器 ····· 427

MEM ····· 427

MySQL产品支持 ····· 431

MySQL企业版的使用 ····· 431

安装 ····· 432

修复监控代理问题 ····· 433

监控 ····· 434

查询分析器 ····· 440

更多信息 ····· 443

小结 ····· 443

第三部分 高可用性环境

第14章 云计算解决方案 ····· 447

什么是云计算 ····· 448

云架构 ····· 450

云计算是一种经济的选择吗 ····· 453

云计算实例 ····· 454

云计算的好处 ····· 454

云计算供应商 ····· 455

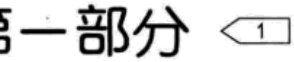
AWS ····· 456

技术简要概述 ····· 457

它是如何工作的 ····· 461

Amazon Cloud工具……461
入门……465
使用磁盘……479
接下来怎么做……484
云中的MySQL……484
MySQL复制和EC2……485
EC2中使用MySQL的最佳实践……488
开源云计算……490
小结……491
第15章 MySQL集群……493
什么是MySQL集群……494
术语和组件……494
MySQL集群和MySQL有何不同……495
典型配置……495
MySQL集群的特点……496
局部和全局冗余……497
日志处理……498
冗余和分布式数据……498
MySQL集群的架构……499
如何存储数据……501
分区……504
事务管理……504
联机操作……505
配置实例……506
入门……507
启动MySQL集群……508
集群测试……513
关闭集群……514

达到高可用性 ····· 514
系统恢复 ····· 517
节点恢复 ····· 518
复制 ····· 518
获得高性能 ····· 523
高性能的注意事项 ····· 523
高性能的最佳实践 ····· 524
小结 ····· 527
附录A 复制建议和窍门 ····· 529
Slave停机了，怎么办 ····· 529
检查冗长的二进制日志 ····· 530
利用复制在表中重建数据 ····· 530
基于语句的日志 ····· 530
基于行的日志 ····· 531
使用MySQL Proxy来完成多Master的复制 ····· 531
使用默认的存储引擎 ····· 532
MySQL Cluster 多源（Multisource）复制 ····· 532
多路（Multichannel）复制故障转移 ····· 533
使用当前数据库来过滤 ····· 533
Slave上的列比Master上多 ····· 534
Slave上的列比Master上少 ····· 535
选择某几列复制到Slave ····· 536
复制心跳 ····· 537
在环形复制中忽略服务器 ····· 538
功能预览：延时复制 ····· 538
功能预览：脚本式复制 ····· 539
功能预览：Oracle算法 ····· 540
索引 ····· 541

第一部分 

复制

要提供强大的 MySQL 环境，首要的任务是建立复制。今后在高可用性其他方面的配置和管理中，这里所学的东西将大有帮助。

第1章 引言

为了找到一份新工作，Joel仔细浏览了分类广告。虽然目前他的工作已经很好了，且自从他进入大学以来，公司就待他不薄，但是已经毕业很多年，他希望在职业生涯中能有更多的挑战。

“这个看上去不错。”他边说边在一个招聘MySQL计算机专家的广告上画了个圈。他有MySQL经验，显然也符合这份工作的学历要求。在快速浏览了其他几个广告后，他决定打电话询问关于MySQL的工作。问完几个粗略的问题之后，人力资源部经理让他两天后面试。

经过两天的三轮面试以后，他被引见给公司的董事长兼首席执行官，Robert Summerson，进行最后一轮的技术面试。在提问过程中，当Summerson先生停下来参看关于Joel的相关记录时，他在旁边等着。到目前为止，大多是些关于信息技术的普通问题，但Joel知道关于MySQL的刁难问题就要来了。

终于，面试官说：“Thomas先生，我对你的回答印象很深刻。我可以叫你Joel吗？”

“可以，先生。”Joel说。面试官第三次看了他的记录，Joel又是一阵不自在。

“告诉我你对MySQL的了解。”Summerson先生把手放在桌子上，凝视着Joel说道。

Joel开始解释他所知道的MySQL，倒腾着前一天晚上看的一大堆资料。十分钟后他要说的都说完了。

Summerson先生沉默了几分钟后站起来，并向Joel伸出一只手。当Joel起身与Summerson先生握手时，Summerson说：“这就是我想听的东西，Joel。这份工作

是你的了。”

“谢谢您，先生。”

4 Summerson 先生示意 Joel 随他一起走出办公室。“我带你去人力资源部，这样我们可以把你加在公司员工的工资名单上。你可以从星期一开始为期两周的试用吗？”

Joel 非常高兴，情不自禁地笑了：“是的，先生。”

“很好。” Summerson 先生再次与 Joel 握手，说道：“我希望你来的时候已经做好评估我们的 MySQL 服务器配置的准备。我需要一份关于服务器配置和健康状况的完整的报告。”

Joel 开车出停车位的时候，从之前的兴奋中冷静了下来。他并没有立即回家，而是去了最近的书店。“我需要一本 MySQL 的好书。”他想。

现在，你已经决定要进行大规模安装工作，并对其进行维护。很好，你将会迎来一段非常有趣且有益的时光。

与运行一个小站点相比，支撑一个大企业需要计划、远见、经验，甚至更周密的计划。作为一个大型企业的数据库管理员，你可能需要做以下的事情：

- 提供灾难发生时核心业务数据的恢复计划。理论上说这个过程至少需要执行一次。
- 通过采集大量用户数据并监控网站各节点的负载，提供优化计划。
- 当用户数量急剧增长时的快速横向扩展计划。

对于所有这些情况，提前计划并准备好必要时的快速应对是很重要的。

鉴于不是所有使用大数据集的应用服务器都是网站，我们倾向于使用“部署（*deployment*）”这个术语，而不是站点（*site*）或网站（*website*），来表示用于支持某种应用的服务器。它可能是一个站点，也可能只是一个 CRM（客户关系管理）系统或者一个在线游戏。本书侧重于该系统的数据库层，但也有一些应用层和数据库层的整合例子。

为了保证站点可响应和可用，你需要做两件事情：系统的数据备份和冗余。备份可以将节点恢复到它崩溃之前的状态，而冗余则保证即使在一个或更多的节点停止提供服务的情况下，站点仍能继续运行。

有多种方法可以实现备份，方法的选择取决于你的需求。你需要即时恢复到一个精确的时间点吗？如果是，就必须保证拥有执行即时恢复（point-in-time-recovery，PITR）必

需的条件。你想在备份的同时保持服务器正常运行吗？如果是，就必须保证正在使用的是某种形式的在线备份方法。

冗余是通过硬件副本来实现的，让几个实例并行运行，并通过复制（Replication）在几个机器上保存相同数据的多个可用副本。如果其中一个机器失效，可以切换到另一个拥有相同数据副本的机器。

和复制一样，备份也在系统扩展和需要时添加新节点方面起了重要作用。打个比方，如果使用正确，甚至可能按下按钮就自动地添加新的从节点。

到底什么是复制

阅读本书时你可能已经对复制了如指掌了。尽管如此，这里还是要介绍一下概念和意图。

复制就是复制一个服务器上（称为主节点服务器或者简称主节点）的所有改变到另一个服务器（称为从节点服务器或简称从节点）。复制通常用来创建主节点的一个可靠副本，另外复制也可能用于其他用途。

两种最常见的使用复制的例子是：(1) 创建一个主节点的备份，以避免主节点崩溃时丢失数据；(2) 拥有一份主节点的副本，从而在不干扰其他业务的情况下执行报表和分析工作。

对于一个小型企业，很多事情变得更加简单，但可能要做更多复制的工作，如下所示。

- 支持多个工作场所
 可能需要在每个地点维护服务器并将改变复制到其他工作场所，从而使得信息处处可用。这就可能有必要保护数据，同时也要满足一些合法的需求，从而保证用于审计目的的业务信息可用。
- 即使有一个服务器停机，也能保证业务的持续运行
 如果原始服务器失效，其他服务器也可以处理所有的访问量。
- 即使有灾难发生，也能保证业务持续运行
 复制可以将数据变化发送给不同地理位置的其他数据中心。
- 错误保护（“oops”）
 将一个从节点连到主节点，可能从节点总是比主节点落后一个固定的时间周期（例如一个小时），这样就会产生一个延迟的从节点（Delayed Slave）。如果这时主节点上发生错误，可以找到出错语句并在从节点执行之前删除它。

目前很多应用程序中使用复制的两个最重要的应用之一就是横向扩展（scale out）。现今的应用程序通常是读密集型的，具有高读写比。为了减少主节点上的负载，你可以搭建

一个从节点只用于响应读请求。通过一个负载均衡器，可以将读请求定向到合适的从节点，而写请求则交给主节点处理。

在横向扩展的场景下使用复制时，理解MySQL复制的异步性很重要，即事务首先在主节点上提交，然后复制给从节点并在从节点上应用。这意味着主节点和从节点可能并不一致，而且如果复制持续运行，从节点将会落后于主节点。

使用异步复制的好处在于它比同步复制更快、更具可扩展性，但在那些实时数据很重要的情况下，必须采用同步的方式以保证信息总是最新的。

6 复制的另一个重要应用是通过添加冗余来保证高可用性。最常见的技术是使用双主配置（*dual-master setup*），即通过复制使得一对主节点总是可用，每个主节点都是对方的镜像。如果其中一个主节点失效，另一个会立即接手。

除了双主配置，还有其他获得高可用性的技术，如使用共享或复制磁盘。尽管它们与MySQL不特别相关，但这些技术对于保证高可用性来说也是很重要的工具。

那么，是否需要备份

备份策略是保持系统可用的一个关键部分。常规的服务器备份提供了应对崩溃和灾难的安全措施，它们某种程度上可以通过复制来实现。然而，即使正确并高效地使用了复制技术，还是有一些复制无法解决的问题。你需要有一个切实可行的备份策略来应对以下的情况：

- 错误保护

 如果发现错误，一般在它实际发生以后很长时间才发现，这时复制便不再有效。在这种情况下，必须把系统回滚到错误发生前的时刻并解决问题。这需要一个切实可行的备份计划。

 如果你正在使用有延迟的从节点，复制可以提供一些错误保护；但如果在某个延迟时间段内发现错误，从节点可能已经接受了这个错误改变。所以，一般来说，仅仅使用复制不可能做到完全的错误保护，还需要备份。

7 - 建立新服务器

 当建立新的服务器时——用于横向扩展的从节点或者备用的新的主节点，都需要对现有服务器做备份并在新服务器上恢复这个备份映像。这需要有一个快速高效的备份方法来最小化宕机时间，并保持系统负载在一个可接受的水平。

- 法律原因

 除了纯粹业务原因需要保护数据外，可能还有法律规定要求保证数据安全，即使在

灾难发生时。不遵守这些规定会给业务运作带来重大问题。

简而言之，不管有没有其他的预防措施来保证数据的安全，备份策略对于业务运作都是必需的。

什么是监控

即使已经正确搭建了复制，还需要理解系统负载，密切监控可能发生的任何问题。客户使用模式变化导致业务需求变化，需要平衡系统以尽可能高效地使用资源，并减少由于资源利用的突发改变而失去可用性的风险。

要应对这些变化，监控、度量和计划的方法很多，比如：

- 为频繁读取的表添加索引。
- 重写查询或者改变数据库的结构，以加速执行时间。
- 如果锁被占用了很长时间，则表示多个连接正在使用同一个表。可能要切换存储引擎。
- 在横向扩展的数据库复制架构下，如果某些从节点处理了极大量的查询，处于过热状态，系统可能需要重新均衡以保证所有从节点都被平均地访问。
- 为了解决资源使用的突发变化，需要确定每个服务器的正常负载，并了解系统何时由于负载的突然增加而响应变慢。

不监控就没有办法观察到有问题的查询、过热的从节点或者使用不恰当的表等。

还有什么我可以阅读的

有大量的文献讲述如何使用 MySQL 完成各种各样的工作，同样也有许多关于高可用性系统的著作。

如果你打算从事 MySQL 工作，下面是我们强烈推荐的书籍清单：

"*MySQL*"，*Paul DuBois* (*Addison-Wesley*)

> 这是 MySQL 的参考书，一共 1200 页（真的！），涵盖了一切关于 MySQL 你想知道的东西（可能也有许多你不想知道的）。

"*High Performance MySQL, Second Edition*"，*Baron Schwartz, et al.* (*O'Reilly, http://oreilly.com/catalog/9780596101718/*)

> 这是企业级 MySQL 应用的最佳书籍之一，包括如何优化查询，以及保证系统的响应性和可用性。

"*Scalable Internet Architectures*", *Theo Schlossnagle* (*Sams Publishing*)

作者是行业内最杰出的思想家之一，本书是从事系统扩展的人员的必读本。



本书使用 Python 库（名为 *MySQL Python Replicant*）开发管理性任务。MySQL Python 可以从 *https://launchpad.net/mysql-replicant-python* 下载。

小结

下一章我们将开始介绍复制的基础知识，所以找个舒服的凳子，打开电脑，然后开始吧……

乔尔正在调整椅子，这时有人敲门。

"还习惯吗，Joel？" Summerson 先生问道。

Joel 不知道该说些什么。第一天上班他就接到任务要建立一个复制从节点。尽管他花的时间比预期要长得多，但还是得看老板的态度。Joel 首先想到的回答是："是的，先生，我还在试图解决这把椅子。"

"文档写得不错，Joel。我想让你写一份报告，解释一下你认为应该如何改善我们的数据库服务器管理。"

Joel 点点头："没问题。"

"好。我再多给你一天的时间整理办公室。我希望周三下班之前看到报告。"

Joel 还没来得及回答，Summerson 先生就走开了。

9 Joel 坐了下来，然后转动椅子上的另一个手柄。"咔嚓"一声，好像靠背掉了，迫使他甩开双臂。"哇！"他看了一下门口，笨拙地摆好椅子，还好没有人看到他的即兴体操。"好了，这下手柄彻底坏了。"他说。

第2章 MySQL复制原理

一阵急促的敲门声把Joel吓了一跳，又是老板来骚扰了。Joel还没来得及说“请进”，老板就已经走进来，说道：“Joel，有人抱怨说我们的响应时间变慢了，想想怎么让它加速。管理员告诉我应用程序的读操作太多了，看看能不能减少一些负载。”

Joel还没来得及回答，Summerson先生就出去了。“我想他的意思是我们需要一个更强大的服务器。”Joel想。

好像知道Joel的想法一样，Summerson先生又折回来进门说：“哦，顺便说一句，开业时我们买了所有机器，有一些服务器还没用过，你看看怎么处理。OK，Joel？”说完他就走了。

“我在想我到底能不能搞定。”Joel边想边将他心爱的MySQL书从书架上拿下来，迅速浏览了目录，找到“复制”这一章，断定这就是他想要的。

如果正确使用MySQL复制，那么它就是一个非常有用的工具；但如果复制出现故障，或配置和使用不当，就会相当令人头疼。本章从一个简单的配置开始，涵盖MySQL复制的基础内容，然后介绍一些基本技巧以丰富你的“复制工具箱”。

本章包括以下使用案例。

- 通过热备份来避免灾难
 如果服务器宕机，一切都将停止：不能执行（可能很关键的）事务，无法得到用户信息，也不能检索其他重要数据。要不惜一切代价避免这种情况发生，因为它会严重破坏业务。最简单的方法就是配置一个额外的服务器专门作为热备份（hot standby），在

Master 宕机的时候随时接管任务。

12

- 产生报表

 直接用服务器上的数据创建报表将大大降低服务器的性能，在某些情况下尤其显著。如果产生报表需要大量的后台作业，最好创建一个额外的服务器来运行这些作业。停止报表数据库上的复制，然后在不影响主要业务服务器的情况下运行大量查询，从而得到数据库在某一特定时间的快照。例如，如果在每天最后一个事务处理完毕后停止复制，可以提取日报表而其他业务仍正常运转。

- 调试和审计

 还可以审查服务器上的查询。例如，查看某些查询是否有性能问题，以及服务器是否由于某个糟糕的查询而不同步。

复制的基本步骤

本章将介绍一些最大化复制的效率和价值的尖端技术。首先需要建立一个如图 2-1 所示的简单复制，即单一的主从复制实例。这里不需要了解内部架构或复制过程的执行细节(在后面介绍更复杂的方案之前，我们再探讨这些内容)。

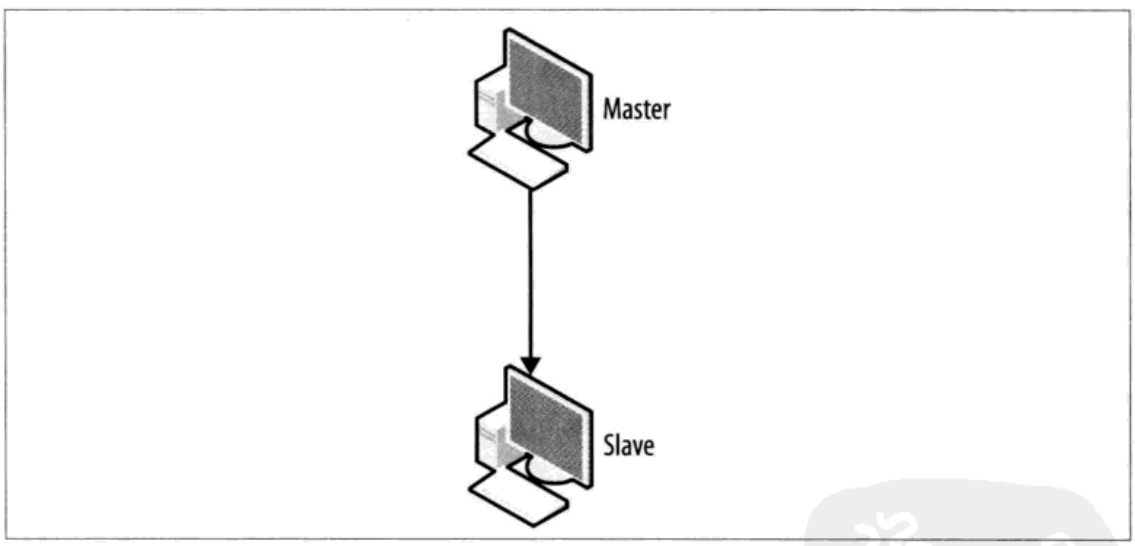


图2-1：简单的复制

建立基本的复制可以总结为以下三个简单步骤：

1. 配置一个服务器作为 Master。

2. 配置一个服务器作为 Slave。

3. 将 Slave 连接到 Master。

除非你从一开始就计划复制且 *my.cnf* 文件中的配置正确，否则步骤 1 和步骤 2 要求必须重启每个服务器。



执行这些步骤最简单的方法是拥有一个可以修改 *my.cnf* 文件权限的 shell 账号（通常是指 mysql 权限）和一个具有 ALL 权限的账号。

必须严格限制在生产环境中授予权限。准确的指导方法请查阅后面将要介绍的“配置复制的权限”。

配置Master

将服务器配置为 Master，要确保该服务器有一个活动的二进制日志（*binary log*）和唯一的服务器 ID。我们后面再仔细研究二进制日志，现在只要知道二进制日志上保存了 Master 上的所有改变，并且可以在 Slave 上重新执行即可。服务器 ID 用于区分服务器。要创建二进制日志和服务器 ID，你需要停掉服务器，然后按例 2-1 所示将 `log-bin`，`log-bin-index`，和 `server-id` 选项添加到 *my.cnf* 配置文件。黑体字部分为添加的配置选项。

例2-1：将配置选项添加到my.cnf以配置Master

```
[mysqld]
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
log-bin         = master-bin
log-bin-index   = master-bin.index
server-id       = 1
```

`log-bin` 选项给出了二进制日志产生的所有文件的基本名（稍后你将看到，二进制日志包含多个文件）。如果你创建了一个以 `log-bin` 为扩展名的文件名，该扩展名将被忽略，而只使用文件的基本名。

`log-bin-index` 选项给出了二进制索引文件的文件名，这个索引文件保存了所有 binlog 文件的列表。

严格地说，不需要为 `log-bin` 选项提供值，其默认值是 `hostname-bin`。`hostname` 的值来自 `pid-file` 选项，默认值是主机名（可以通过 `gethostname(2)` 系统调用得到）。如果

14 管理员后来修改了主机名，binlog 文件名也会随之改变，但是索引文件仍可以获取正确的值。最好为服务器创建一个机器无关的唯一的服务器名，因为一系列 binlog 文件中途改名可能会很混乱。

如果没有为 `log-bin-index` 赋予任何值，其默认值与 binlog 文件的基本名相同（如果没有为 `log-bin` 提供值，则默认为 `hostname-bin`）。也就是说，如果你不赋值给 `log-bin-index`，索引文件名会随主机名的改变而改变。所以，如果你改变主机名然后重启服务器，将找不到索引文件，从而认为索引文件不存在，导致二进制日志为空。

每个服务器都有一个唯一的服务器 ID，所以如果一个 Slave 连接了 Master，并且其 `server-id` 的参数值与 Master 相同，则会产生 Master 和 Slave 服务器 ID 相同的错误。

将 `log-bin` 和 `server-id` 选项添加到配置文件后重启服务器，然后添加一个复制用户，这样便完成了配置。

修改 Master 的配置文件以后，重启 Master，使配置生效。

Slave 启动一个标准的客户端连到 Master，并请求 Master 将所有的改动转储给它。Slave 连接时要求 Master 上有一个特殊复制权限的用户。例 2-2 展示了 Master 上的一个标准的 `mysql` 客户端会话，通过命令创建新用户并赋予适当的权限。

例2-2：*在Master上创建一个复制用户*

```
master> CREATE USER repl_user;
Query OK, 0 rows affected (0.00 sec)
master> GRANT REPLICATION SLAVE ON *.*
    -> TO repl_user IDENTIFIED BY 'xyzzy';
Query OK, 0 rows affected (0.00 sec)
```



`REPLICATION SLAVE` 权限并没有什么特别之处，只是这个用户能够从 Master 上取得二进制日志的转储数据。完全可以给一个常规用户赋予 `REPLICATION SLAVE` 权限，但最好还是将复制 Slave 用户与其他用户区别开来。 这样的话，以后如果想禁止某些 Slave 的连接，只要删除该用户就可以了。

15 配置Slave

配置完 Master 后，还需要配置 Slave。与 Master 一样，需要为每个 Slave 分配一个唯一的服务器 ID。考虑使用 `relay-log` 和 `relay-log-index` 选项向 *my.cnf* 文件添加中继日志文件（relay log file）和中继日志索引文件（relay log index file）的文件名（我们将在后面的“复制架构基础”中详细讨论中继日志）。例 2-3 给出了推荐的配置选项，其中新添加的选项加粗表示。

例2-3：添加选项到my.cnf文件来配置Slave

```
[mysqld]
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
server-id       = 2
relay-log-index = slave-relay-bin.index
relay-log       = slave-relay-bin
```

与 log-bin 和 log-bin-index 选项一样，relay-log 和 relay-log-index 选项的默认值取决于 hostname。relay-log 的默认值是 hostname-relay-bin，relay-log-index 的默认值是 hostname-relay-bin.index。使用默认值有个问题，即一旦服务器的主机名改变，将会因为无法找到中继日志索引文件而认为中继日志文件为空。

修改 *my.cnf* 文件以后，重启 Slave，使配置生效。

连接Master和Slave

现在创建基本的复制只剩下最后一步了：将 Slave 指向 Master，让它知道从哪里进行复制。为此你需要知道 Master 的 4 部分信息：

- 主机名
- 端口号
- Master 上拥有 REPLICATION SLAVE 权限的用户账号
- 该用户的密码

配置 Master 时已经创建了一个拥有正确权限的用户账号及密码。主机名由操作系统确定，不能通过 *my.cnf* 配置，但是端口号可以通过 *my.cnf* 来分配（如果没有指定端口号，将使用默认值 3306）。最后两步对创建和运行复制来说是必需的，使用 CHANGE MASTER TO 命令将 Slave 指向 Master，然后使用 START SLAVE 命令启动复制。

```
slave> CHANGE MASTER TO
    ->    MASTER_HOST = 'master-1',
    ->    MASTER_PORT = 3306,
    ->    MASTER_USER = 'repl_user',
    ->    MASTER_PASSWORD = 'xyzzy';
Query OK, 0 rows affected (0.00 sec)

slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

恭喜！你已经建立了 Master 和 Slave 之间的第一个复制。如果你在 Master 的数据库上做一些改动，例如创建新表并填充数据，将发现这些改变都会复制到 Slave。试试吧！建立一个测试数据库（如果还没有的话），再创建一些表，然后添加一些数据到表中，看看这些改变是否会复制到 Slave 中。

注意 MASTER_HOST 参数的值可以是主机名或 IP 地址。如果是主机名，则可以通过调用 gethostname(3) 得到对应的 IP 地址，即通过域名查找来解析主机名，其结果与配置有关。配置域名解析的步骤不在本书讨论的范围内。

配置复制的权限

要把 Slave 连接到 Master 做复制，除了要一个能访问关键文件的 shell 账户，还要有一个具有特定权限的账户。出于安全原因，通常要严格控制配置 Master 和 Slave 的账户只拥有必需的权限。

- CREATE USER 权限用于创建和删除用户。
- REPLICATION SLAVE 权限用于复制账户，且附有 GRANT OPTION。

想要执行进一步的复制相关的过程（在本章后面将说明），还需要更多选项：

- 执行 FLUSH LOGS 命令（或任何 FLUSH 命令）需要 RELOAD 权限。
- 执行 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令需要 SUPER 或 REPLICATION CLIENT 权限。
- 执行 CHANGE MASTER TO 命令需要 SUPER 权限。

例如，赋予足够的权限给 mats 用户来执行本章中的所有过程，使用如下命令：

```
server> GRANT REPLICATION SLAVE, RELOAD, CREATE USER, SUPER
    -> ON *.*
    -> TO mats@'192.168.2.%'
    -> WITH GRANT OPTION;
```

17

二进制日志简介

复制过程需要二进制日志（或者 *binlog*），它记录了服务器数据库上的所有改变。你需要理解二进制日志是如何控制复制过程或解决问题的，所以本节我们将介绍一些背景知识。

图 2-2 展示了复制的结构示意图，其中包括 Master 及其二进制日志和 Slave，该 Slave 通过二进制日志获取 Master 上的改变。我们将在第 6 章详细阐述复制的架构。语句执行结束时，将在二进制日志的末尾写入一条记录，同时通知语句解析器语句已经执行完毕。

通常只有即将执行完毕的语句才会被写入二进制日志，但是在一些特殊情况下其他信息也会被写入，以对语句进行添加或替换。不久你就会知道这么做的原因，但是目前我们假设只有执行的语句才会写入二进制日志。

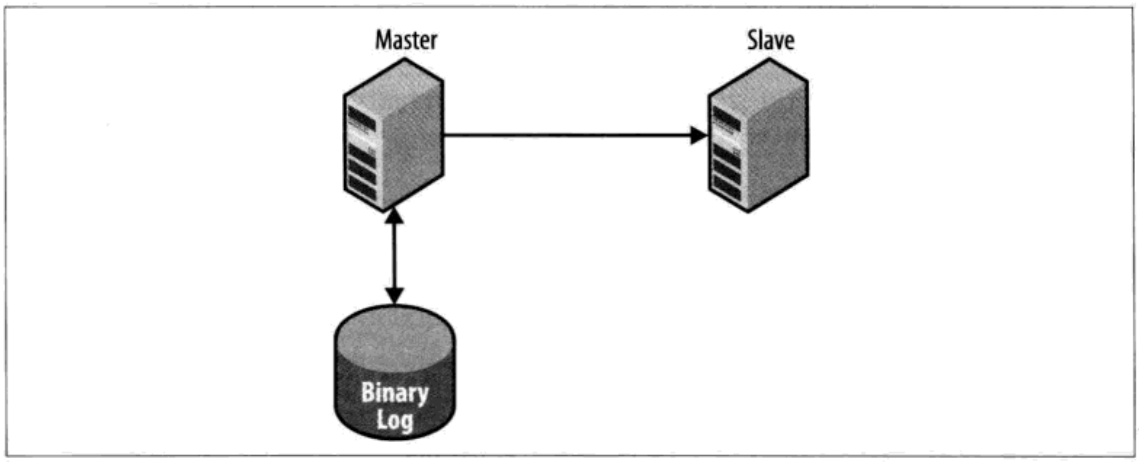


图2-2：二进制日志在复制中的作用

二进制日志记录了什么

二进制日志的目的是记录数据库中表的更改，然后用于复制和PITR（将在第12章中讨论），另外少数审计情况下也会用到。

注意二进制日志只包括数据库的改动，所以对那些不改变数据的语句则不会写入二进制日志。

传统意义上说，MySQL复制记录了产生变化的SQL语句，称为基于语句的复制（*statement-based replication*）。基于语句的复制的缺点是无法保证所有语句都正确复制。 18
所以在5.1版本中，MySQL还提供了基于行的复制（*row-based replication*）。相比基于语句的复制，基于行的复制将每一次改动记为二进制日志中的一行。基于行的复制不仅更加方便，而且有时候速度更快。

考虑一个含有多表连接或WHERE条件的复杂更新，看看这两种复制有什么不同。你真正需要知道的是更新后每行的状态，而不是像基于语句的复制那样把所有的逻辑都在Slave上重新执行。相反，如果某个更新改变了10 000行，你可能宁愿只记录这个语句，而不是像基于行的复制那样去记录10 000个单独的改动。

我们将在第6章涉及基于行的复制，介绍它的实现和使用。下面的例子重点讲解基于语句的复制，这样更容易理解数据库的执行。

观察复制的动作

用上一节中的复制例子，我们来看看一些简单语句的 binlog 事件。先使用命令行客户端连接 Master，然后执行一些命令获取二进制日志：

```
master> CREATE TABLE tbl (text TEXT);
Query OK, 0 rows affected (0.04 sec)

master> INSERT INTO tbl VALUES ("Yeah! Replication!");
Query OK, 1 row affected (0.00 sec)

master> SELECT * FROM tbl;
+--------------------+
| text               |
+--------------------+
| Yeah! Replication! |
+--------------------+
1 row in set (0.00 sec)

master> FLUSH LOGS;
Query OK, 0 rows affected (0.28 sec)
```

FLUSH LOGS 命令强制轮换(rotate)二进制日志，从而得到一个“完整的”二进制日志文件。使用 SHOW BINLOG EVENTS 命令进一步查看该文件，如例 2-4 所示。

例2-4：检查二进制日志里有哪些事件

```
master> SHOW BINLOG EVENTS\G
*************************** 1. row ***************************
   Log_name: master-bin.000001
        Pos: 4
 Event_type: Format_desc
  Server_id: 1
End_log_pos: 106
       Info: Server ver: 5.1.33, Binlog ver: 4
19 *************************** 2. row ***************************
   Log_name: master-bin.000001
        Pos: 106
 Event_type: Query
  Server_id: 1
End_log_pos: 197
       Info: use `test`; CREATE TABLE tbl (text TEXT)
*************************** 3. row ***************************
   Log_name: master-bin.000001
        Pos: 197
 Event_type: Query
  Server_id: 1
End_log_pos: 305
       Info: use `test`; INSERT INTO tbl VALUES ("Yeah! Replication!")
*************************** 4. row ***************************
```

```
   Log_name: master-bin.000001
        Pos: 305
 Event_type: Rotate
  Server_id: 1
End_log_pos: 349
       Info: master-bin.000002;pos=4
4 rows in set (0.02 sec)
```

这个二进制日志包含 4 个事件：一个格式描述事件、两个查询事件，以及一个日志轮换（rotate）事件。查询事件是如何将数据库上执行的语句写入二进制语句，而格式描述事件和日志轮换事件则用于服务器内部管理二进制日志。第 6 章将详细讨论这些事件，这里我们简单看一下每个事件所包含的字段：

- `Event_type`
 这是事件的类型。这里我们已经看到了三种类型，但还有很多类型。事件类型是给 Slave 传送信息的基本方法。目前 MySQL 5.1.18 到 5.1.39 版本一共有 27 种事件类型（其中有些事件是不使用的，但是为了向后兼容而保留了）。这个范围是可扩展的，如果以后的版本需要其他事件类型，可以添加新的事件类型。
- `Server_id`
 这是创建事件的服务器的 ID。
- `Log_name`
 这是用来存储事件的文件名。一个事件只能存储在一个文件中，永远不能跨两个文件。
- `Pos`
 这是事件在文件中的开始位置，即事件的第一个字节。
- `End_log_pos`
 这是事件在文件中的结束位置，也是下一个事件的开始位置。这个位置比事件的最后一个字节高一位，因此事件的字节范围为 `Pos` 到 `End_log_pos - 1`。通过计算 `End_log_pos - Pos` 可以得到这个事件的长度。
- `Info`
 这是事件信息的可读文本。不同的事件会显示不同的信息，不过至少可以通过查询事件来知道它所包含的语句。

前两个字段 `Log_name` 和 `Pos` 组成了事件的二进制日志位置（*binlog position*），用于标识事件的地点或位置。除了以上字段以外，每个事件还包括很多其他信息，例如时间戳，即从纪元（用经典的 UNIX 时间格式表示，例如 1970-01-01 00:00:00 UTC）开始的秒数。

二进制日志的结构和内容

前面讲过，二进制日志并不是一个单独的文件，而是由一系列易于管理的（例如在不影

响新日志的情况下移除旧的日志）文件组成的。二进制日志包括一组存储实际内容的二进制日志文件和一个二进制日志索引文件，而二进制索引文件包含所有使用的二进制日志文件的文件名，它用来跟踪存在的二进制日志文件。 图 2-3 展示了一个二进制日志的结构。

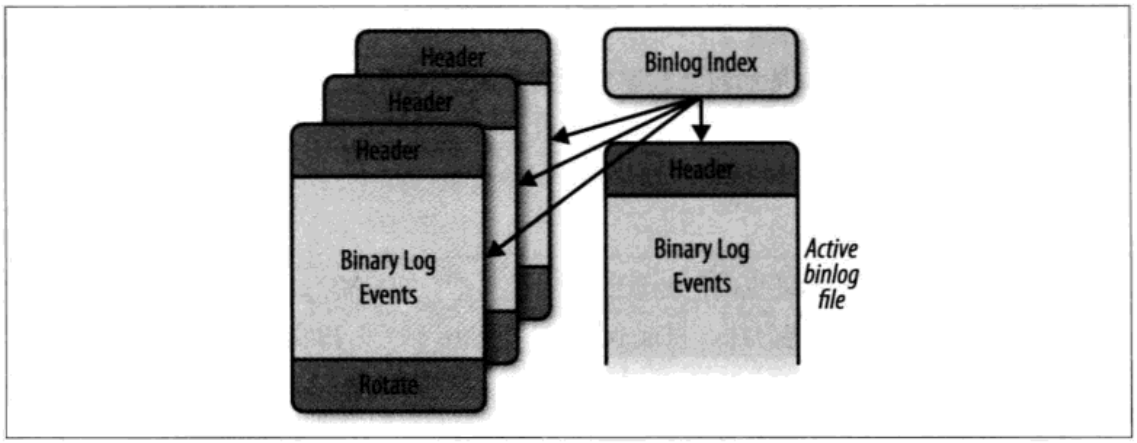


图2-3：二进制日志的结构

其中一个二进制日志文件是活动二进制日志文件（*active binlog file*），即当前正在被写入的文件（通常也从这个文件读取）。

每个二进制日志文件都以格式描述事件（*format description event*）开始，以日志轮换事件（*rotate event*）结束。格式描述日志事件包括产生该文件的服务器版本号、服务器信息及二进制日志信息等。日志轮换事件包含下一个二进制日志文件的名称，以告知二进制日志继续写入哪个文件。

21

每个二进制日志文件有多个二进制日志事件，各个事件之间相互独立，同时也是构成二进制日志的基本单位。格式描述日志事件还有一个标记，标记二进制日志文件是否正常关闭。如果正在写入二进制日志文件，则设置该标记；如果文件关闭，则清除该标记。这样就可以检测出在崩溃事件中损坏的二进制日志文件，并允许通过复制进行恢复。

如果在 Master 上执行其他语句，你会发现一件奇怪的事情——在二进制日志中看不到变化：

```
master> INSERT INTO tbl VALUES ("What's up?");
Query OK, 1 row affected (0.00 sec)

master> SELECT * FROM tbl;
+--------------------+
| text |
```

```
+--------------------+
| Yeah! Replication! |
| What's up?         |
+--------------------+
1 row in set (0.00 sec)

master> SHOW BINLOG EVENTS\G
same as before
```

这些新事件是怎么回事？我们已经知道，二进制日志由若干文件组成，而 SHOW BINLOG EVENTS 语句只显示第一个二进制日志文件的内容。这与大多数用户的期望相反，他们想看的是活动二进制日志文件。 如果第一个二进制日志文件名是 *master-bin.000001*（这个文件包含前面显示的事件），你可以使用下面的命令查看下一个二进制日志文件（本例该文件名为 *master-bin.000002*）的事件：

```
master> SHOW BINLOG EVENTS IN 'master-bin.000002'\G
*************************** 1. row ***************************
   Log_name: master-bin.000002
        Pos: 4
 Event_type: Format_desc
  Server_id: 1
End_log_pos: 106
       Info: Server ver: 5.1.30-log, Binlog ver: 4
*************************** 2. row ***************************
   Log_name: master-bin.000002
        Pos: 106
 Event_type: Query
  Server_id: 1
End_log_pos: 205
       Info: use `test`; INSERT INTO tbl VALUES("What's up?")
2 rows in set (0.00 sec)
```

你可能已经注意到在例 2-4 中，二进制日志以日志轮换事件结尾，Info 字段包含下一个二进制日志文件名和事件的开始位置。使用 SHOW MASTER STATUS 命令查看当前正在写入的是哪个二进制日志文件： <22>

```
master> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: master-bin.000002
        Position: 205
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

查看二进制日志以后，停止并重置 Slave，然后删除表：

```
slave> DROP TABLE tbl;
Query OK, 0 rows affected (0.00 sec)
```

```
slave> STOP SLAVE;
Query OK, 0 rows affected (0.08 sec)

slave> RESET SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

接着，删除表，然后重置 Master 刷新：

```
master> DROP TABLE tbl;
Query OK, 0 rows affected (0.00 sec)

master> RESET MASTER;
Query OK, 0 rows affected (0.04 sec)
```

RESET MASTER 命令删除了所有二进制日志文件并清空了二进制日志索引文件。RESET SLAVE 命令删除了 Slave 复制所用的所有文件，重新开始。



无论是 RESET MASTER 还是 RESET SLAVE 都是在运行复制时有效。因此：

- 执行 RESET MASTER 命令（在 Master 上）时，确保没有 Slave 连接到该 Master。
- 执行 RESET SLAVE 命令（在 Slave 上）时，先执行 STOP SLAVE 命令，确保 Slave 上没有活动的复制。

本章涵盖了大多数基本事件，对于全部事件的详细信息请参考 MySQL 内部手册。

23 使用Python管理复制

对于处理大规模部署来说，管理过程的自动化很重要。那么你可能会自问："如果我们可以使过程自动化，那样不是很好么？"如果真的这样，你会很高兴自己做到了。使用前面章节的描述，这里我们将设计一个简单的程序库来管理复制。接下来的几章中我们会扩展该程序库加入新功能。Launchpad 上有这个项目，可以从上面找到信息或下载源代码和文档。

首先必须创建一个关于如何通过复制连接服务器的模型。连接大量服务器的方法很多，但是连接时需要以一定的配置来建立它们，这就叫拓扑（topology）。我们将在第 5 章介绍拓扑。一个简单的基础拓扑如图 2-4 所示，这是一个树形拓扑，有两个 Master（用来提供高可用性）。

如图 2-4 所示，其基本思想是建立一个服务器如何连接到计算机（任何计算机，比如笔记本）的模型，并设计程序库，通过改变模型来管理连接。例如，将 Slave 重新连接到另一个 Master，只需要在模型里重连接 Slave，程序库会发送适当的命令来执行该任务。

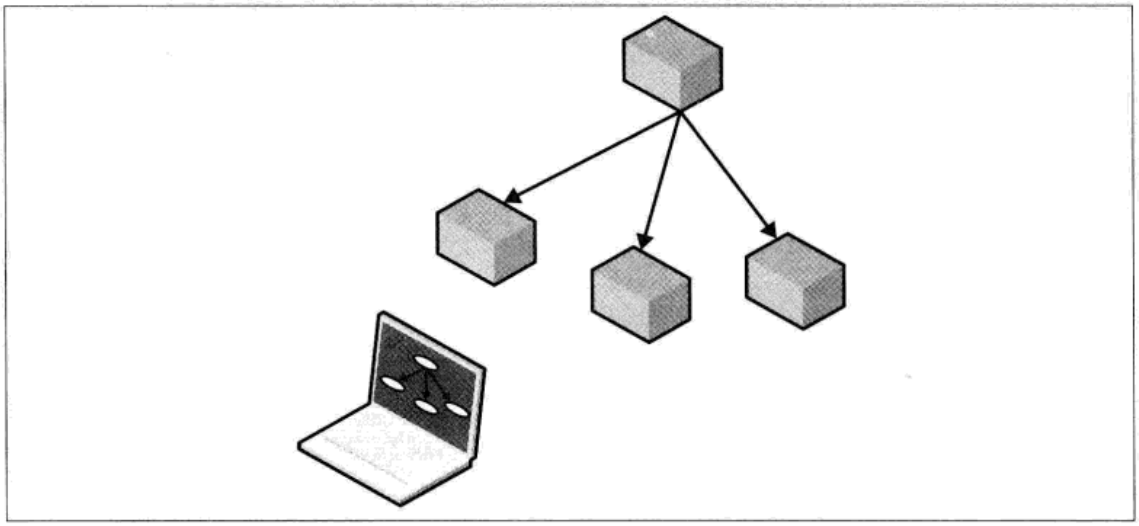


图2-4：一个复制拓扑模型

为了让程序库能跨平台使用，也为了部署的多样性，需要注意以下几点：

- 服务器可能会在多种操作系统上运行，例如 Windows，Linux，以及不同版本的 UNIX，如 Solaris 或 Mac OS X。启动和停止服务器的过程，以及配置文件的文件名，都因使用的操作系统不同而各有差异。因此这个程序库必须支持不同的操作系统，同时还能够扩展到不在程序库里的新操作系统上使用。
- 部署服务器运行的可能是不同版本的 MySQL。例如将部署升级到新版本时，其中将包含新老版本的混合物。程序库要能够处理这样的部署。
- 部署中的服务器角色不同，所以该程序库应该可以为服务器指定不同的角色。此外，程序库还可以创建新角色。
- 每个服务器都可以执行 SQL 查询。这是配置的要求，同时也是获取部署管理所需的信息的要求。
- 每个机器都可以执行 shell 命令。这是执行 SQL 接口无法完成的管理任务的要求。
- 应该能够对服务器配置文件中的选项进行添加和删除。
- 该程序库支持在一个机器上部署多个服务器。这就需要能够识别单个机器上不同 MySQL 服务器的配置文件和数据库文件。
- 需要一套实用程序来执行像建立复制这样的普通任务，同时还可以添加新的实用程序函数，用以扩展程序库。

接口隐藏了尽可能多的复杂性，提供了一个简单的 Python 接口。作者们之所以选择 Python，是因为它简明、易读，可用于所有运行 MySQL 的操作系统，而且在通用脚本中越来越流行。例 2-5 通过一个简短的例子将所有的 Slave 重定向到新的 Master，说明了如何使用程序库。

注意这段代码只是关于如何使用程序库的简单例子。下面的代码如果运行在活动的服务器上，将会停止复制，而且可能导致事务丢失。读者将在第 4 章中看到如何正确地更改 Master。

例2-5：使用程序库来重定向Slave

```
import MyDeployment
for slave in MyDeployment.slaves:
    slave.stop()
    change_master(slave, MyDeployment.master[1])
    slave.start()
```

25

下面几节给出了应用所需的代码。为了避免不必要的语言混乱，我们已经去掉了一些错误检查，以及其他保证稳定和安全的防御性措施。你可以在这里查到完整的程序库代码：

http://launchpad.net/mysql-replicant-python.

基本类及函数

使用这个程序库首先需要了解一些常用参数的基本定义。

第一类是程序库函数使用的异常。

- `Error`
 这是程序库中所有异常的基类。
- `EmptyRowError`
 当一个查询试图查找一个字段而没有返回任何行的时候，将抛出该异常。
- `NoOptionError`
 当 `ConfigManager` 没有找到相应选项的时候，将抛出该异常。
- `SlaveNotRunningError`
 当希望 Slave 运行但 Slave 却不在运行时，将抛出该异常。
- `NotMasterError`
 因服务器不是 Master 而操作不合法时，将抛出该异常。
- `NotSlaveError`
 因服务器不是 Slave 而操作不合法时，将抛出该异常。
- `Position`
 该类表示二进制日志的位置，由文件名和文件内的字节偏移量组成。该类提供了一个表示方法，将二进制日志位置以一种可解析的表示方式输出（如将它们存到另一个地方或只是观察其值）。
 为了对这些位置进行比较和排序，该类定义了比较运算符使程序库按顺序存放二进制日志的位置。注意，不同服务器上的位置可能不同，所以不同服务器之间的位置

比较没有意义。

- User

 这个类用名字和密码来表示用户，用于多种账户类型，如 MySQL 用户账户，shell 用户账户，以及复制用户（我们将在后面介绍）。

操作系统

使用一组类抽象出不同的操作系统之间的差异。其思想是为不同操作系统的不同任务实现提供一个类方法。这样我们只需要停止和启动服务器的方法即可。

- Machine

 该类是机器的基类，保存了此类机器的公共信息。一个机器实例至少拥有如下成员：

 - Machine.defaults_file

 my.cnf 文件在机器上的默认位置。

 - Machine.start_server(server)

 启动服务器的方法。

 - Machine.stop_server(server)

 停止服务器的方法。

- Linux

 Linux 机器上负责服务器运行的类，使用存储在 */etc/init.d* 的 init(8) 脚本来启动和停止服务器。

- Solaris

 Solaris 机器上负责服务器运行的类，使用 svadm(1M) 命令来启动和停止服务器。

服务器类

该类定义了对外接口中高级函数使用的所有原始函数：

- Server.Server(*name*, ...)

 Server 类代表系统中的服务器。整个系统中，每个运行中的服务器都对应一个服务器对象。这里只描述了最重要的参数，要想了解全部参数，请查阅 Launchpad 上的项目（project）页面。

 - name

 这是服务器的名字，为 pid-file，log-bin 和 log-bin-index 选项提供值。如果没有提供 name 参数，可以从 pid-file，log-bin 和 log-bin-index 选项中推导得到，或者最起码可以使用默认值。

- host，port 和 socket

 host 是服务器所在的主机，port 是 MySQL 客户端连接服务器的端口号，socket 是该连接使用的套接字（同一主机上）。

27 - ssh_user

 连接服务器所需的用户名和密码。用于执行管理命令，如启动和关闭服务器以及读写配置文件。

- sql_user

 MySQL 用户账户连接服务器所需的用户名和密码，用于执行 SQL 命令。

- machine

 包含操作系统原语的对象。使用这个名字是为了避免与标准程序库中的 os 模块发生冲突。这个参数允许使用不同的技术来启动和关闭服务器。还有其他任务以及操作系统相关的参数将在后面介绍。

- server_id

 可选参数，保存服务器的 ID，在每个服务器的配置文件中定义。如果没有此项，将从 Slave 的配置文件读取服务器 ID。如果配置文件中也没有服务器 ID，则说明该服务器没有参与复制，既不是 Master 也不是 Slave。

- config_manager

 可选参数，保存对配置管理器的引用，用于查询服务器的配置信息。

- Server.connect() 和 Server.disconnect()

 connect 和 disconnect 方法分别用于某个会话执行命令前建立服务器连接，以及会话完成后断开连接。

 这些方法很有用，因为有时候即使 SQL 命令已经执行完毕，也需要保持服务器的连接状态。否则，举个例子，执行 FLUSH TABLES WITH READ LOCK 命令时，如果连接被删除，则锁也会自动释放。

- Server.ssh(*command*) 和 Server.sql(*command, args*)

 用于在服务器上执行 shell 命令或 SQL 命令。

ssh 和 sql 方法都会返回一个迭代器 (Iterator)。ssh 返回已执行命令的输出行列表，而 sql 返回内部类 Row 的对象列表。Row 类定义了__iter__和 next methods 方法，用于返回迭代的行。例如：

```
for row in server.sql("SHOW DATABASES"):
    print row["Database"]
```

28 为了处理只返回一行的语句，这个类还定义了_getitem_方法，用于从单行中读取字段，如果没有记录则抛出异常。也就是说，如果你知道返回值只有一行（从许多 SQL 语句都可以知道这点），就不用像前面例子那样循环了，而是直接写成：

```
print server.sql("SHOW MASTER STATUS")["Position"]
```

- Server.fetch_config() 和 Server.replace_config()

 fetch_config 和 replace_config 方法将配置文件从远程服务器读取到内存中，从而允许用户增加或删除选项，或者改变某些选项的值。例如，使用以下方法为 log-bin 和 log-bin-index 选项赋值：

```
config = master.fetch_config()
config.set('log-bin', 'capulet-bin')
config.set('log-bin-index', 'capulet-bin.index')
master.replace_config(config)
```

- Server.start() 和 Server.stop()

 start() 和 stop() 方法用于将信息发给 machine 对象执行，其中 machine 对象与服务器使用的操作系统有关。这两个方法分别用于启动或关闭服务器。

服务器角色

不同角色的服务器的工作方式略有不同。例如，Master 需要一个复制用户用于 Slave 连接，而 Slave 并不需要这样的用户账户，除非它同时也是 Master 而且有其他 Slave 连接到它。为了保证服务器配置的灵活性，引入几个类用来描绘不同的角色（*role*）。

在服务器上使用 imbue 方法，向服务器发送恰当的命令，以正确配置角色。注意在部署的生命周期中服务器可能会转换角色，所以这里指的是初始部署时配置的角色。无论如何，服务器总有一个部署角色和一个关联角色。

如果服务器转换角色，可能需要删除某些服务器的配置信息，因此还定义了 unimbue 方法专门用于服务器的角色切换。

这个例子中只定义了三个角色，在后面的章节中读者将看到更多角色的定义。

- Role

 这是所有角色的基类。每个派生类都需要定义 imbue 方法和可选的 unimbue 方法，允许单个服务器注入这个角色。Role 类还定义了很多辅助函数帮助派生类完成一些常见任务。 29

 - Role.imbue(server)

 通过执行适当的代码，将新角色注入给服务器。

 - Role.unimbue(server)

 允许某个角色在注入其他角色之前执行清理操作。

 - Role._set_server_id(server, config)

如果没有配置服务器 ID，则使用该方法将服务器 ID 设为 server.server_id.。如果配置了服务器 ID，则将 server.server_id 的值设为这个服务器 ID。

- Role._create_repl_user(server, user)
 该方法用于在服务器上创建复制用户，并授予它复制 Slave 必需的权限。
- Role._enable_binlog(server, config)
 通过设置 log-bin 和 log-bin-index 选项来启用服务器上的二进制日志。如果服务器的 log-bin 选项已经有值，则这个方法将什么都不做。
- Role._disable_binlog(server, config)
 通过清除配置文件中的 log-bin 和 log-bin-index 选项来禁用二进制日志。

- Vagabond
 这是分配给那些不参与复制的服务器的默认角色。因此，如果服务器为“vagabond”，那么它没有任何职责。
- Master
 用做 Master 的服务器角色。该角色设置服务器 ID，启用二进制日志，并为 Slave 创建复制用户。复制用户的用户名和密码存储在服务器上，所以 Slave 连接后可以查询复制用户名。
- Final
 这是（最终的）Slave（即没有自己的二进制日志的 Slave）的角色。如果要将 Final 角色注入服务器，需要提供服务器 ID，禁用其二进制日志，然后发出 CHANGE MASTER 命令将 Slave 连接到 Master 上。

注意修改配置文件之前要停止服务器，并在修改完成后重启服务器。只有启动服务器时需要读取配置文件，并在读取完毕后关闭。不过为了安全起见，修改配置文件之前最好先停止服务器。

30 这里有一个关键性的设计，即不存储角色对应的服务器的任何状态信息。为角色对象保存一份所有 Master 的列表或许不错，但是由于在部署的整个生命周期中服务器的角色是不断变化的，所以角色仅用于建立系统。角色可以包含参数，可以使用参数对多个服务器进行相同的配置。

```
slave_role = Final(master=MyDeployment.master)
for slave in MyDeployment.slaves:
    slave_role.imbue(slave)
```

建立新Slave

既然已经了解了二进制日志，下面我们要解决前面建立 Slave 的过程中存在的一个基本问题。前面配置 Slave 时，并没有说明复制从哪里开始，所以 Slave 将从头开始读取 Master 上的二进制日志。如果 Master 已经运行了一段时间，这显然不是个好方法：不仅

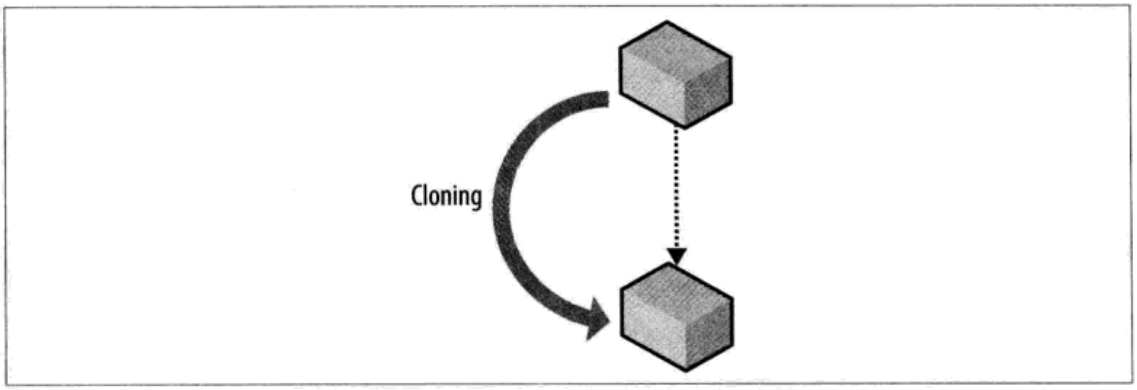


图2-5：克隆Master来创建新Slave

一旦数据库被锁定，就可以创建备份，并记录 binlog 位置了。由于 Master 没有任何改动，SHOW MASTER STATUS 命令将正确返回当前二进制日志文件及其 binlog 位置。我们将在第 6 章详细讨论 SHOW MASTER STATUS 和 SHOW MASTER LOGS 命令。

```
master> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: master-bin.000042
        Position: 456552
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

32

将要写入的下一个事件的位置是 master-bin.000042，456552，这就是复制的起点，这个位置点之前的所有东西都在备份里。记下 binlog 位置后就可以创建备份了。创建数据库备份最简单的方法是用 mysqldump：

```
$ mysqldump --all-databases --host=master-1 >backup.sql
```

有了可靠的 Master 副本，就可以为数据库中的表解锁，允许数据库继续处理查询。

```
master> UNLOCK TABLES;
Query OK, 0 rows affected (0.23 sec)
```

接下来，在 Slave 上使用 mysql 实用程序恢复备份：

```
$ mysql --host=slave-1 <backup.sql
```

已经在 Slave 上恢复了 Master 的备份，现在可以启动 Slave 了。利用前面记下的 Master 的 binlog 位置，使用 CHANGE MASTER TO 命令配置 Slave，然后启动 Slave。

```
slave> CHANGE MASTER TO
    ->   MASTER_HOST = 'master-1',
    ->   MASTER_PORT = 3306,
    ->   MASTER_USER = 'slave-1',
```

```
    ->   MASTER_PASSWORD = 'xyzzy',
    ->   MASTER_LOG_FILE = 'master-bin.000042',
    ->   MASTER_LOG_POS = 456552;
Query OK, 0 rows affected (0.00 sec)

slave> START SLAVE;
Query OK, 0 rows affected (0.25 sec)
```



mysqldump 可以自动执行前面的步骤。要对 Master 服务器上的所有数据库进行逻辑备份，输入：

```
$ mysqldump --host=master -all-databases \
> --master-data=1 >backup-source.sql
```

--master-data=1 选项使 mysqldump 写 CHANGE MASTER TO 语句，且参数为二进制日志文件及其位置（可通过 SHOW MASTER STATUS 得到）。

然后就可以在 Slave 上恢复备份了：

```
$ mysql --host=slave-1 <backup-source.sql
```

注意，只能使用 --master-data=1 执行 CHANGE MASTER TO 语句。后面克隆 Slave 的时候，需要执行下一节的所有步骤。

恭喜！你已经克隆了 Master，并建立和运行了一个新 Slave。根据 Master 的负载情况，你可能需要 Slave 从前面记录的位置开始运行，这比从头开始容易得多。

根据备份需要的时间长短不同，可能有大量的数据需要同步，所以，在将 Slave 联机（online）之前，首先要仔细阅读后面将要介绍的“数据的一致性管理”。

克隆Slave

只要有一个 Slave 连在 Master 上，就可以使用这个 Slave 创建新的 Slave，而不需要再离线（offline）Master 了。如果数据库很大或流量较高，那么停机时间可能会相当长，因为既要考虑创建备份的时间又要考虑 Slave 同步的时间。

克隆 Slave 的过程如图 2-6 所示，与克隆 Master 基本相同，区别在于如何找到 binlog 位置。另外，注意你克隆的那个 Slave 同时还在执行从 Master 的复制。

首先，必须在备份前停止 Slave，保证 Slave 上不再有变化发生。如果创建备份时复制仍在进行，而在备份期间数据库又发生了变化，这时就会得到不一致的备份映像。但是，如果使用某种在线备份方法，如 InnoDB Hot Backup（InnoDB 的热备份工具），则不需要在创建备份前停止 Slave。

```
original-slave> STOP SLAVE;
Query OK, 0 rows affected (0.20 sec)
```

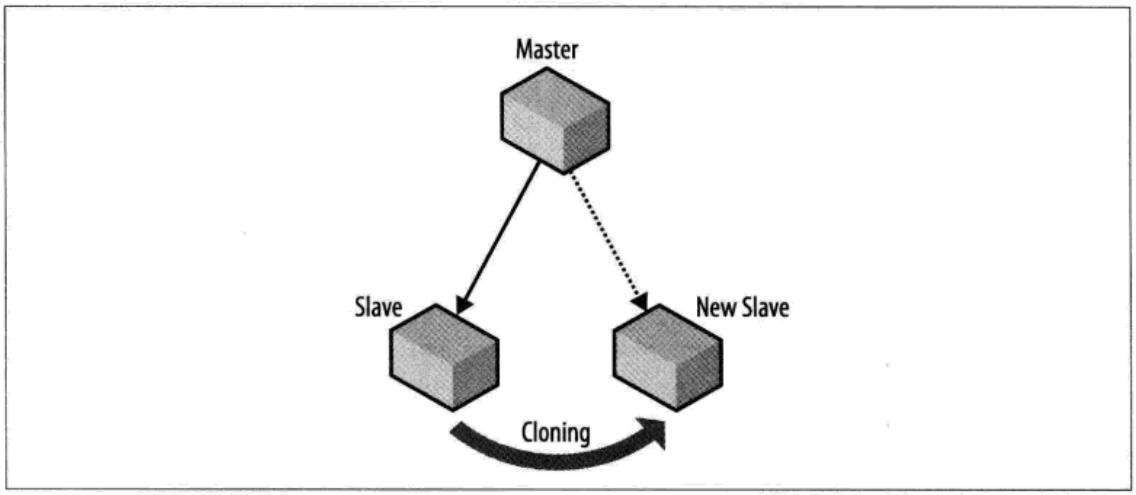


图2-6：克隆Slave以创建一个新的Slave

34 Slave停止以后，就可以像从前一样刷新（flush）数据表，然后创建备份。创建了Slave的备份（而不是Master的备份）后，使用SHOW SLAVE STATUS命令（而不是SHOW MASTER STATUS）来确定从哪里开始复制。该命令的输出很多，第6章将详细介绍。如果要获得Master二进制日志中Slave即将执行的下一个事件的位置，请注意Relay_Master_Log_File和Exec_Master_Log_Pos字段的值。

```
original-slave> SHOW SLAVE STATUS\G
   ...
Relay_Master_Log_File: master-bin.000042
   ...
 Exec_Master_Log_Pos: 546632
```

创建了备份然后在新Slave上恢复以后，将复制配置为从这个位置开始，然后启动新的Slave：

```
new-slave> CHANGE MASTER TO
        ->   MASTER_HOST = 'master-1',
        ->   MASTER_PORT = 3306,
        ->   MASTER_USER = 'slave-1',
        ->   MASTER_PASSWORD = 'xyzzy',
        ->   MASTER_LOG_FILE = 'master-bin.000042',
        ->   MASTER_LOG_POS = 546632;
Query OK, 0 rows affected (0.19 sec)

new-slave> START SLAVE;
Query OK, 0 rows affected (0.24 sec)
```

克隆Master和克隆Slave只有一些细节上的区别，这意味着我们的Python程序库可以

将这两个过程合并起来，即在源服务器上创建备份从而创建新的 Slave，然后将这个新 Slave 连接到 Master。



创建备份的常见方法是调用 FLUSH TABLES WITH READ LOCK，然后归档数据库文件。通常这样很快，但是在 *InnoDB* 中使用 FLUSH TABLES WITH READ LOCK 是不安全的！

FLUSH TABLES WITH READ LOCK 会锁定表，这样就不会产生任何新事务，但后台仍然有一些活动在继续进行。

使用下面的方法安全地创建 InnoDB 数据表的备份：

1. 关闭服务器，然后复制文件。如果数据库很大，最好采取这种方法，因为这时使用 mysqldump 进行数据恢复会很慢。
2. 执行 FLUSH TABLES WITH READ LOCK（前面已经介绍过）之后，使用 mysqldump。
3. 执行 FLUSH TABLES WITH READ LOCK 之后，使用快照方法，如 LVM（Linux 平台），ZFS（Zettabyte File System）快照（Solaris）等。

克隆操作的脚本

Python 程序库实现克隆 Master 的方法很简单，使用一个 Server 对象代表 Master，然后从这个 Master 上复制数据库。使用 clone 函数实现，见例 2-7。

克隆 Slave 的过程类似，但它是在一台服务器上创建备份，然后新 Slave 连接另一台服务器进行复制。同时支持 Master 和 Slave 的克隆操作很容易，只需要使用两个参数：source 参数指定从哪里创建备份，以及 use_master 参数指出备份恢复后 Slave 需要连接的服务器。这样调用 clone 方法：

```
clone(slave = slave[1], source = slave[0], use_master = master)
```

下面要写一些实用工具函数来实现克隆功能，这些函数在其他地方也可以派上用场。例 2-6 用到了如下函数：

- fetch_master_pos
 从 Master 获取 binlog 位置（即 Master 即将写入二进制日志的下一个事件的位置）。
- fetch_slave_pos
 从 Slave 获取 binlog 位置（即从 Master 读取的下一个事件的位置）。
- replicate_from
 需要提供的参数包括 Slave，Master 及 binlog 位置，将 Slave 定向到 Master，并从给定的 binlog 位置开始复制。

replicate_from 函数从 Master 上读取 repl_user 字段获取复制用户的用户名和密码。但 Server 类的定义中没有该字段，它是服务器注入 Master 角色时添加的字段。

例2-6：获取服务器的master和slave位置的功能函数

```
    _CHANGE_MASTER_TO = """CHANGE MASTER TO
       MASTER_HOST=%s, MASTER_PORT=%s,
       MASTER_USER=%s, MASTER_PASSWORD=%s,
       MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s"""

    def replicate_from(slave, master, position):
        slave.sql(_CHANGE_MASTER_TO, (master.host, master.port,
                                      master.repl_user.name,
                                      master.repl_user.passwd,
                                      position.file, position.pos))

    def fetch_master_pos(server):
        result = server.sql("SHOW MASTER STATUS")
        return mysqlrep.Position(server.server_id, result["File"], result["Position"])

36  def fetch_slave_pos(server):
        result = server.sql("SHOW SLAVE STATUS")
        return mysqlrep.Position(server.server_id, result["Relay_Master_Log_File"],
                                 result["Exec_Master_Log_Pos"])
```

这些都是创建 clone 函数所需要的函数。为了克隆 Slave，应用程序传递一个单独的 use_master 参数，让 clone 函数将新 Slave 定向到该 Master 做复制。为了克隆 Master，应用程序忽略这个 use_master 参数，让 clone 函数使用“源”服务器作为 Master。

创建服务器备份的方法很多，例 2-7 仅选择了一种方法，即使用 mysqldump 创建服务器的逻辑备份。稍后我们将演示如何将这个备份过程一般化，使得无论采用任何备份方法都可以使用相同的基本代码建立新 Slave。

例 2-7：克隆Master或Slave的函数

```
def clone(slave, source, use_master = None):
    from subprocess import call
    backup_file = open(server.host + "-backup.sql", "w+")
    if master is not None:
        stop_slave(source)
    lock_database(source)
    if master is None:
        position = fetch_master_position(source)
    else:
        position = fetch_slave_position(source)
    call(["mysqldump", "--all-databases", "--host='%s'" % source.host],
         stdout=backup_file)
    if master is not None:
        start_slave(source)
```

```
    backup_file.seek()                  # Rewind to beginning
    call(["mysql", "--host='%s'" % slave.host], stdin=backup_file)
    if master is None:
        replicate_from(slave, source, position)
    else:
        replicate_from(slave, master, position)
    start_slave(slave)
```

执行常见的复制任务

每个常见的横向扩展（scale out）策略，如热备份（hot standby）等，都有其各自的实现细节和可能的陷阱。下面将向读者展示如何执行某些任务，以及如何使 Python 程序库支持这些任务。



本节中的例子省略了密码。配置某些控制服务器的账户时，可以只允许某些控制部署的特定主机访问（通过创建类似 `mats@'192.168.2.136'` 这样的账户），或者也可以给这些命令提供密码。

37

报表

大多数企业需要大量的例行报告：已售物品的周报表，开支和收入的月报表，以及大量的数据挖掘报表，以发现某些趋势或为营销部门识别重点群体。

在 Master 上运行这些查询会很麻烦。数据挖掘查询可能需要大量的计算资源、拖慢正常操作，而结果却发现，比如说，那个左撇子重点群体可能并不值得这么做。此外，这些报表通常不是很紧急（与处理日常事务相比），所以没有必要尽快创建报表。换句话说，这些报表对时间要求不严格，即使一个小时不够，花两个小时完成也没什么关系。

报表往往需要精确的间隔时间，如汇总当天所有的销售额，需要在适当的时候停止复制，而不至于将第二天的销售额也计入报表。因为无法在特定日期或时间的事件发生时停止 Slave，所以必须得想其他的方法。

最好是重新利用备用服务器（一个或两个，如果报表需求很多，可以用两个），然后将它设置为从 Master 复制。做报表的时候停止复制，运行报表程序，然后再重新启动复制，这样所有的操作都不会影响到 Master。

假设每天做一次报表，包括所有从午夜到午夜的交易。午夜时停止报表 Slave，使得午夜后 Slave 上不再有事件执行而午夜前所有事件都已经执行。我们不想手动执行此操作，考虑如何使这个过程自动化，按照以下步骤执行：

1. 午夜前，可能是午夜前 5 分钟，停止报表 Slave，这样就不会再从 Master 接收事件。
2. 午夜后，检查 Master 上的二进制日志，找到午夜前记录的最后一个事件。显然，如果在午夜之前进行这一步操作，可能无法得到当天的所有事件。
3. 记录该事件的 binlog 位置，然后启动 Slave。
4. Slave 运行，直到到达这个位置后停止。

38 第一个问题是如何正确地调度任务。调度任务的方法很多，与操作系统有关。这里我们不会详细讨论这个问题，但你可以在后面的“UNIX 中的任务调度”中看到如何在类 UNIX 操作系统（如 Linux）中调度任务。

停止 Slave 很简单，执行 STOP SLAVE 命令，然后记录 binlog 位置。

```
slave> STOP SLAVE;
Query OK, 0 rows affected (0.25 sec)

slave> SHOW SLAVE STATUS\G
   ...
Relay_Master_Log_File: capulet-bin.000004
   ...
  Exec_Master_Log_Pos: 2456
1 row in set (0.00 sec)
```

其他三个步骤要在实际报告启动前执行，通常作为生成实际报告的脚本的一部分。描述脚本之前，先考虑如何执行各个步骤。

调用 mysqlbinlog 实用工具读取的二进制日志的内容。这一点后面会详细介绍，第二步中使用了这个工具。mysqlbinlog 实用工具提供两个选项用于部分读取二进制日志，即 --start-datetime 和 --stop-datetime。因此，要获得停止 Slave 时刻到午夜期间的所有事件，可以使用以下命令：

```
$ mysqlbinlog --force --read-from-remote-server --host=reporting.bigcorp.com \
> --start-datetime='2009-09-25 23:55:00' --stop-datetime='2009-09-25 23:59:
59' \
> binlog files
```



事件中存储的时间戳是指语句开始执行的时间戳，而不是它写入二进制日志的时间戳。

--stop-datetime 选项会在其 date/time 值后的第一个时间戳时刻停止产生事件，所以可能存在这样的事件，在 date/time 之前开始执行，但却被写入了二进制日志。该事件不在给定的范围内。

这时 Master 正在写二进制日志，需要提供 --force 选项。否则，mysqlbinlog 无法读取二进制日志。这个命令需要读取一组 binlog 文件。由于这些文件的名称取决于配置选项，所以只能从 Slave 上获取这些文件名。然后，搞清楚 mysqlbinlog 命令需要哪些 binlog 文件。使用 SHOW BINARY LOGS 命令可以很容易地获取 binlog 文件名列表：


```
master> SHOW BINARY LOGS;
+--------------------+-----------+
| Log_name           | File_size |
+--------------------+-----------+
| capulet-bin.000001 |     24316 |
| capulet-bin.000002 |      1565 |
| capulet-bin.000003 |       125 |
| capulet-bin.000004 |      2749 |
+--------------------+-----------+
4 rows in set (0.00 sec)
```

本例只有四个文件，其实可以有更多。在停止 Slave 前扫描这么多文件只是浪费时间，所以最好能减少需要读取的文件数量，以便于找到正确的停止位置。由于第 1 步中已经记下了 binlog 位置，Slave 停止运行时，很容易找到这时的 binlog 文件名，然后将该文件名及其后面所有的文件名作为 mysqlbinlog 实用工具的输入。通常只有一个文件（或两个，例如在关闭和启动报表事件期间二进制日志发生了轮换）。

如果只用几个 binlog 文件执行 mysqlbinlog 命令，将得到事件相关信息的文本输出。

```
$ mysqlbinlog --force --read-from-remote-server --host=reporting.bigcorp.com \
>    --start-datetime='2009-09-25 23:55:00' --stop-datetime='2009-09-25
     23:59:59' \
>    capulet-bin.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#090909 22:16:25 server id 1 end_log_pos 106 Start: binlog v 4, server v...
ROLLBACK/*!*/;
    .
    .
    .
# at 2495
#090929 23:58:36 server id 1 end_log_pos 2650 Query thread_id=27 exe...
SET TIMESTAMP=1254213690/*!*/;
SET /*!*/;
INSERT INTO message_board(user, message)
    VALUES ('mats@sun.com', 'Midnight, and I'm bored')
/*!*/;
```

有趣的是，最后一个事件的 end_log_pos 参数（本例中该值为 2650），就是午夜后的下

一个事件即将被写入的位置。

40 如果注意看前面的命令输出，你会发现没有任何关于这个字节位置指向哪个 binlog 文件的相关信息。mysqlbinlog 需要指定一个文件来查找事件。如果向 mysqlbinlog 命令提供单个文件，那么这个文件名是显而易见的，但如果提供两个文件，则必须确定当天最后一个事件在哪个文件里。

观察含有 end_log_pos 的那行，还会看到事件类型。每个 binlog 文件的开始都是格式描述事件（此类事件的输出在前面），检查这些事件，以确定要查找的事件的位置。如果输出两个格式描述事件，则该事件在第二个文件中；如果只有一个格式描述事件，则在第一个文件中。

报表工作开始前的最后一步是启动复制，然后在正确的位置停止它，即午夜后事件写入的位置（即将写入或已被写入）。可以使用鲜为人知的命令 START SLAVE UNTIL 来完成。该命令的参数包括 Master 日志文件和 Slave 停止时对应的 Master 日志位置。Slave 到达给定位置时自动停止。

```
report> START SLAVE UNTIL
    -> MASTER_LOG_POS='capulet-bin.000004',
    -> MASTER_LOG_POS=2650;
Query OK, 0 rows affected (0.18 sec)
```

与 STOP SLAVE 命令一样（不含 UNTIL），START SLAVE UNTIL 命令也会立即返回，但是如果 Slave 已到达应该停止的位置就不立即返回了。所以，只要 Slave 还在运行，在 STOP SLAVE UNTIL 之后发出的命令仍会继续执行。使用 MASTER_POS_WAIT 函数等待 Slave 到达某个停止位置。Slave 到达指定位置之前该函数会阻塞。

```
report> SELECT MASTER_POS_WAIT('capulet-bin.000004', 2650);
Query OK, 0 rows affected (231.15 sec)
```

这时 Slave 已经在当天最后一个事件处停止了，报表过程可以开始分析数据和生成报告了。

使用Python处理报表

使用 Python 进行自动化报表处理是很简单的。

例 2-8 所示的代码是在正确的时间停止报表。

fetch_remote_binlog 函数通过 mysqlbinlog 命令从远程服务器读取二进制日志。将文件内容作为迭代器返回，遍历文件中的各行。还可以提供一个文件列表以优化读取。还可以传递开始的日期 / 时间和结束的日期 / 时间，从而限制结果的日期 / 时间范围。这些

都会传给 mysqlbinlog 程序。

find_datetime_position 函数逐行扫描 binlog 直到找到最后的 end_log_pos，同时对观察到的开始事件进行计数。该函数还联系报表服务器确定何时停止读取 binlog 文件，然后联系 Master 获得 binlog 文件，并找到合适的 binlog 文件开始扫描。

例2-8：运行复制到一个特定日期时间的Python代码 41

```
def fetch_remote_binlog(server, binlog_files=None,
                        start_datetime=None, stop_datetime=None):
    from subprocess import Popen, PIPE
    if not binlog_files:
        binlog_files = [
            row["Log_name"] for row in server.sql("SHOW BINARY LOGS")]

    command = ["mysqlbinlog",
               "--read-from-remote-server",
               "--force",
               "--host=%s" % (server.host),
               "--user=%s" % (server.sql_user.name)]
    if server.sql_user.passwd:
        command.append("--password=%s" % (server.sql_user.passwd))
    if start_datetime:
        command.append("--start-datetime=%s" % (start_datetime))
    if stop_datetime:
        command.append("--stop-datetime=%s" % (stop_datetime))
    return iter(Popen(command + binlog_files, stdout=PIPE).stdout)

def find_datetime_position(master, report, start_datetime, stop_datetime):
    from itertools import dropwhile
    from mysqlrep import Position
    import re

    all_files = [row["Log_name"] for row in master.sql("SHOW BINARY LOGS")]
    stop_file = report.sql("SHOW SLAVE STATUS")["Relay_Master_Log_File"]
    files = list(dropwhile(lambda file: file != stop_file, all_files))
    lines = fetch_remote_binlog(server, binlog_files=files,
                                start_datetime=start_datetime,
                                stop_datetime=stop_datetime)
    binlog_files = 0
    last_epos = None
    for line in lines:
        m = re.match(r"#\d{6}\s+\d?\d:\d\d:\d\d\s+"
                     r"server id\s+(?P<sid>\d+)\s+"
                     r"end_log_pos\s+(?P<epos>\d+)\s+"
                     r"(?P<type>\w+)", line)
        if m:
            if m.group("type") == "Start":
                binlog_files += 1
            if m.group("type") == "Query":
```

```
            last_epos = m.group("epos")
    return Position(files[binlog_files-1], last_epos)
```

现在你可以使用这些函数在真正执行报表任务之前同步报表服务器。

```
master.connect()
report.connect()
pos = find_datetime_position(master, report,
                             start_datetime="2009-09-14 23:55:00",
                             stop_datetime="2009-09-14 23:59:59")
report.sql("START SLAVE UNTIL MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s",
(pos.file, pos.pos))
report.sql("DO MASTER_POS_WAIT(%s,%s)", (pos.file, pos.pos))
.
.
code for reporting
.
.
```

42

可见，复制工作是非常简单的。这个特殊的例子引入了几个关键概念，我们将在后面讨论横向扩展时用到它们，即如何在正确的时间启动和停止 Slave，如何获取 binlog 的位置信息或使用标准工具实现，以及如何整合得出自动化解决方案以满足特定需要。

在UNIX上调度任务

确保午夜前停止 Slave 并且午夜后启动报表，最简单的方法就是建立一个 *cron*(*8*) 作业，向 Slave 发送停止 Slave 的命令然后启动报表脚本。

例如，下面的 *crontab*(*5*) 可以保证 Slave 在午夜前停止，并且在午夜后运行报表脚本让 Slave 前滚（`roll forward`），比如午夜后五分钟。这里我们假设 `stop_slave` 脚本用于停止 Slave，`daily_report` 脚本用于运行日报表（与上述同步启动）。

```
# stop reporting slave five minutes before midnight, every day
55 23 * * * $HOME/mysql_control/stop_slave

# Run reporting script five minutes after midnight, every day
5 0 * * * $HOME/mysql_control/daily_report
```

假设这些脚本放在一个名为 *reporttab* 的 crontab 文件中，使用下面的命令安装 crontab 文件：

```
$ crontab reporttab
```

在Windows Vista上调度任务

Windows Vista 上的任务调度比旧版本的 Windows 更容易。任务调度器（Task Scheduler）有几个明显改进。在的任务调度器是一个 Microsoft 的管理控制台（Microsoft

Management Console）管理单元，并且与事件查看器（Event Viewer）集成，可以使用事件作为触发器来启动任务。

要在 Windows Vista 上调度任务，通过开始菜单上的管理员文件夹（Administrator's Folder），或者在“运行”中（Windows 键 + R）输入 `taskschd.msc`，打开控制面板，选择事件调度器（Event Scheduler），就可以启动任务调度器了。需响应用户账户控制（UAC）对话框然后继续。

从操作窗口（Action pane）中选择“创建基本任务”（Create Basic Task），打开创建基本任务向导，指导你按步骤创建一个简单的按时间触发的任务。在向导的第一个窗口，为任务命名，并提供一个可选的描述，然后单击“下一步”。

第二个窗口允许你指定任务的运行频率。这里有很多选项用于控制任务的运行时间：一次性运行，每天，每周，甚至登录或当特定的事件发生时运行。选择后单击“下一步”。

根据你选择的频率，第三个窗口允许你指定任务运行时间的细节（如日期和时间）。配置好触发的时间选项后单击“下一步”。

第四个窗口指定任务事件发生时（启动任务时）的任务或行动。你可以选择启动程序，发送邮件消息或显示给用户一条消息。选好后单击“下一步”进入下一个窗口。

根据上一个窗口中选择的行动，这里可以指定激发任务的响应操作。例如，如果选择的是运行一个应用程序，则你要输入应用程序或脚本的名字、所有参数及此任务的执行目录。

输入了所有这些信息后，单击“下一步”进入最后一个窗口，检查任务设置。如果确定所有设置都是正确的，单击“完成”。也可以单击“上一步”返回前面任何一个页面来进行修改。最后，单击“完成”后，还可以选择打开属性页，允许对任务做其他改动。

小结

本章介绍了 MySQL 复制，包括为什么要使用复制，以及如何建立复制。还了解了二进制日志。下一章将进一步研究二进制日志。

> Joel 刚刚把关于四个 Slave 如何均衡负载的报告和怎样扩展拓扑结构的计划交给了 Summerson 先生。

“做得好，Joel。现在再向我解释一下什么是 Slave。”

Joel 想叹气却又忍住了，然后说：“Slave 是数据库服务器上的数据拷贝，它从一个叫 Master 的原始服务器上获取变化然后进行数据拷贝……”

第3章 二进制日志

“Joel？”

Joel 吓了一跳，赶紧从桌子底下爬出来，差点撞到头。他解释说：“我刚刚在整理电缆。”

Summerson 先生只是点点头，然后很郑重地说：“我需要你研究一个问题，营销人员现在有新的服务器。他们需要将数据回滚到某个特定的时间点。”

“嗯，那要看……”Joel 准备说点什么，他担心是否有系统以前状态的快照。

“我告诉他们你能搞定。”

说完 Summerson 先生就转身走开了。

过了一会儿，一个美丽的女开发者走到他的门前停了下来，说：“他总是这样，不要介意。我们大都把它叫做过路任务。”她笑了，然后做了自我介绍：“我叫 Amy。”

Joel 绕过桌子走到门口，“我叫 Joel。”

片刻尴尬的沉默后，Joel 说：“呃，我还是继续工作吧。”

Amy 笑着说：“一会儿见。”

“只要专注于你必须做成功的事就好。”Joel 想。他走回桌前，找他上周买的那本 MySQL 方面的书。

上一章简单介绍了二进制日志。本章我们将深入细节，详细描述二进制日志的结构，复制事件的格式，以及如何使用 mysqlbinlog 工具来审查和处理二进制日志的内容。

二进制日志记录了数据库的所有改变，使得任何 Slave 都可以执行同样的更新。通常二进制日志保留了所有更改的记录，可以用于审计目的，看看在数据库中发生了什么；还
46 可用于 PITR（即时恢复），即向服务器回放二进制日志，重复执行二进制日志中记录的更新。

二进制日志仅包含可能改变数据库的语句。请注意，那些尚没有但是可能改变数据库的语句也会记录下来。注意那些可能带来变化的语句，如 DROP TABLE IF EXISTS 或 CREATE TABLE IF NOT EXISTS，以及那些不匹配任何行的语句，如带有 WHERE 条件的 DELETE 和 UPDATE 语句。

SELECT 语句一般不会被记录，因为它们不会对数据库做任何改动。然而，什么都有例外。

服务器上的事务通常不是一个接一个顺序执行的，而是交错地并行执行的。为了防止两个事务之间产生冲突导致不一致的结果，服务器要确保事务的执行是顺序化的。这样执行的事务结果相同，就像它们是串行执行的一样，也就是说，执行顺序是固定的，一个接一个执行。

二进制日志按 Master 上的提交顺序记录事务。虽然事务可能在 Master 上交错执行，但每个事务在二进制日志中的顺序是不变的，取决于事务的提交（commit）时间。

二进制日志的结构

从概念上讲，二进制日志是一系列二进制日志事件（又称 binlog 事件，如果没有概念上的混淆也可以简称为事件）。从第 2 章可以知道，实际上二进制日志包含若干个文件，如图 3-1 所示，它们一起构成了二进制日志。

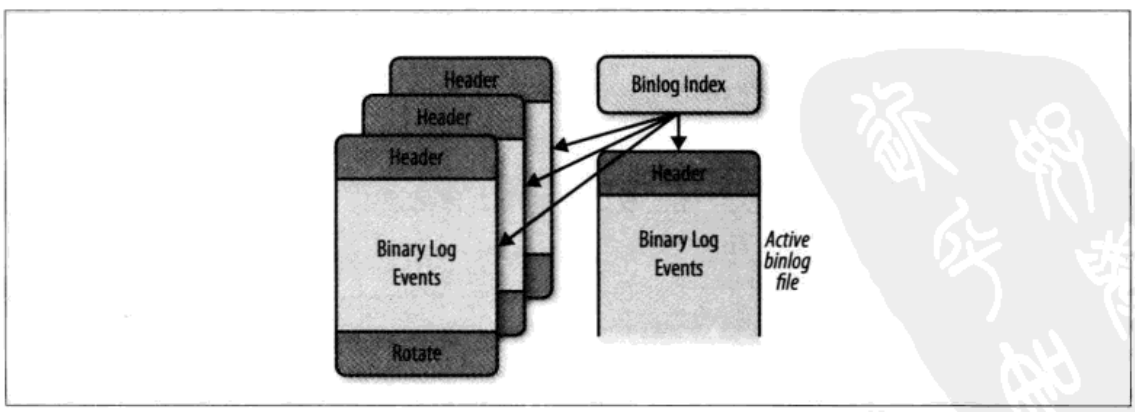


图3-1：二进制日志的结构

真正的事件存储在一系列 *binlog* 文件中，文件名类似于 *host-bin.000001*，还有一个 *binlog* 索引文件，通常文件名为 *host*-bin.index，用来追踪已有的 binlog 文件。当前服务器正在写入的 binlog 文件称为活动（*active*）binlog 文件。如果所有 Slave 都与 Master 同步，那么 Slave 也正在读取该文件。分别使用 `log-bin` 和 `log-bin-index` 选项来控制 binlog 文件及 binlog 索引文件的名字，这将在本章后面讨论。

索引文件跟踪服务器使用的所有 binlog 文件，以便必要的时候服务器能正确地创建新的 binlog 文件。索引文件的每一行都包含了一个 binlog 文件的完整文件名，这个 binlog 文件是二进制日志的一部分。影响 binlog 文件的命令，如 `PURGE BINARY LOGS`，`RESET MASTER`，以及 `FLUSH LOGS`，也同样影响索引文件，使用这些命令添加或删除 binlog 文件会导致索引文件添加或删除行。

如图 3-2 所示，每个 binlog 文件由若干 binlog 事件组成，以 `Format_description` 事件（格式描述事件）作为文件头，以日志轮换事件作为文件尾。注意，如果服务器突然停止或死机，binlog 文件末尾可能不是轮换事件。

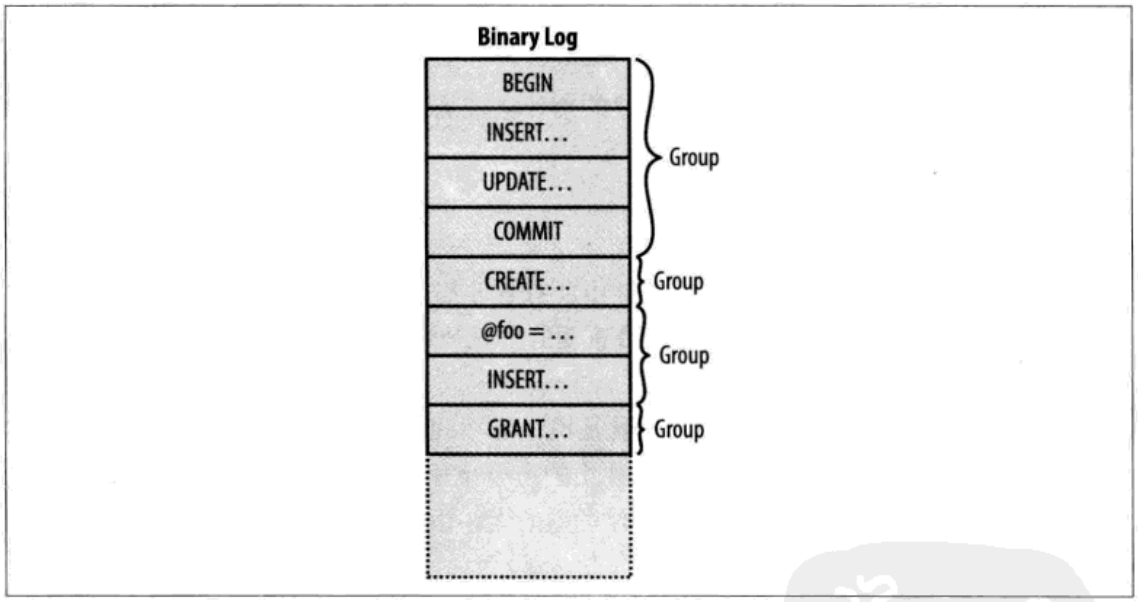


图3-2：含有多组事件的单个binlog文件

`Format_description` 事件包含写 binlog 文件的服务器信息，以及关于文件状态的关键信息。如果服务器关闭或重新启动，会创建一个新的 binlog 文件，同时向其中写入一个新的 `Format_description` 事件。这个事件是必需的，因为在服务器关闭和重启期间可能产生更新。例如，如果升级服务器，需要写入新的 `Format_description` 事件。

服务器写完 binlog 文件后，在文件末尾添加一个轮换事件。该事件包含下一个 binlog 文

件的文件名及其开始读取的位置。

Format_description 事件和轮换事件将在下一节详细描述。

除了 Format_description 和轮换事件之外，binlog 文件中的其他事件都被分成组（*group*）。在事务存储引擎（transactional storage engine）中，每个组大致对应一个事务；但是对于非事务存储引擎，或不属于事务的语句，如 CREATE 或 ALTER 语句，每个语句本身就是一个组。总之，binlog 文件中的事件组要么是不属于事务的单个语句，要么是由多条语句组成的事务。

通常情况下，每个组要么全都执行，要么全都不执行。如果由于某种原因 Slave 在组执行的过程中停机，那么将从该组的起点而不是刚刚执行的语句开始复制。第 6 章将详细描述 Slave 如何执行事件。

Binlog事件的结构

MySQL 5.0 引入了一个新的 binlog 格式，即二进制日志版本 4（binlog format 4）。如果需求增加，以前的格式很难扩展字段，所以二进制日志版本 4 专门设计为可扩展的。5.0 以后版本的服务器仍使用这个事件格式，尽管这些版本服务器的 binlog 格式添加了新事件，且事件中也添加了新字段。本章描述的事件格式是二进制日志版本 4。

每个 binlog 事件由三个部分组成：

- 通用头（*Common header*）
 通用头，顾名思义，就是 binlog 文件中所有事件都具备的信息。
 通用头包含事件的基本信息，其中最重要的字段是事件类型和事件的大小。
- 提交头（*Post header*）
 提交头与特定的事件类型有关。也就是说，对不同的事件类型来说，该字段存储的信息不同。但是，对给定 binlog 文件中的所有事件而言这个头的大小相同，这一点和通用头(Common header)一样。事件类型的大小由 Format_description 事件给出。
- 事件体（*Event body*）
 最后是事件体，其大小可变。事件体的大小在通用头中给出。事件体存储事件的主要数据，因事件类型不同而异。例如，Query 事件的事件体存储查询，而 User_var 事件的事件体则存储某个语句中的用户变量名及其值。

49

所有事件的完整格式列表超出了本书的范围，但由于 Format_description 和轮换事件对解释其他事件至关重要，因此我们将在这里对它们做简要介绍。

前面讲过，Format_description 事件是每个 binlog 文件的开头，描述 binlog 文件中事

件的公共信息，所以不同文件的 Format_description 事件可以不同；这通常发生在服务器升级和重新启动时。

- *binlog* 文件格式版本
 这是 binlog 文件的版本，不要与服务器的版本混淆。MySQL 3.23，4.0 和 4.1 版本使用的是二进制日志版本 3，而 MySQL 5.0 和之后的版本用的是二进制日志版本 4。

 如果开发者对文件或事件的整体结构做了重大改动，则 binlog 文件格式版本会发生变化。5.0 版本 binlog 文件的起始事件采用了新的格式，而且改变了所有事件的通用头，这些都导致了 binlog 文件格式版本的变化。
- 服务器版本
 表示创建文件的服务器的版本字符串，包括服务器的版本及特定的构建信息。其格式通常分为三部分，即版本号、连字符和其他构建选项。例如，“5.1.40-debug-log”表示 5.1.40 版本服务器的调试构建版本。
- 通用头的长度（*common header length*）
 该字段存储了通用头的的长度。这里是指 Format_description 事件，所以不同 binlog 文件该字段的值不同。除了 Format_description 和轮换事件外，其他事件的通用头长度都是可变的。Format_description 事件的通用头长度是不变的，因为任何版本的服务器都要读取这个事件；轮换事件的通用头也是不变的，因为 Slave 连接 Master 时首先要用到轮换事件。所以这两个事件的通用头长度都是不变的，不会随着服务器版本变化。
- 提交头的长度（*post-header lengths*）
 binlog 文件中所有事件的提交头长度都是不变的，该字段存储了各个事件的提交头长度构成的数组。由于不同服务器间的事件数目不同，所以这个字段前面还存储了服务器的事件数目。

由于 Format_description 事件给出了各个事件类型的通用头及提交头的大小，所以不管是添加新事件，还是增加新字段导致提交头的大小增加，都不会影响 binlog 文件的整体格式。 50

每次扩展都要特别小心，确保不会影响对老版本事件的解释。例如，向通用头添加一个字段表明该事件是否被压缩及其压缩类型，但是如果 Slave 从旧版本的 Master 上读取事件而没有这个字段，服务器仍要能够理解这个事件。

记录语句

传统的 MySQL 采用基于语句的复制（statement-based replication）。最近还实现了基于

行的复制（row-based replication），这将在第 6 章讲到。

在基于语句的复制中，实际执行的语句及某些执行信息将一起写入二进制日志，然后在 Slave 上重新执行这些语句。当然并不是所有的语句都会记入日志，要注意有些例外情况。本节描述将语句记入日志的过程，并给出一些重要的说明。

由于二进制日志是公共资源，所有线程都向它写入语句，所以避免两个线程同时更新二进制日志很重要。为此，在事件写二进制日志之前，二进制日志需要获得一个互斥锁 LOCK_log，然后在事件写完成后释放。由于服务器所有的会话线程都向二进制日志写语句，所以这个锁常常会阻塞某些会话线程。

记录数据操作语言

数据操作语言（DML）语句通常是指 DELETE，INSERT 和 UPDATE 语句。为了保证安全地记录日志，MySQL 在获取事务级锁时写二进制日志，然后在二进制日志的写操作完成后释放锁。

表尚未释放锁之前，在语句提交的同时将该语句写入二进制日志，这样就保证了二进制日志始终与语句的更新信息保持一致。如果语句没有写入日志，数据库中的语句与二进制日志中记录的语句产生的数据更新不同，可以“注入”另一个语句填补这种不同。也就是说，语句可以按照不同的顺序记入日志，而不用遵循数据库上的执行顺序，不过，

这样显然会导致 Master 和 Slave 之间不一致。例如，一个带有 WHERE 条件的 UPDATE 语句在 Slave 上更新的行可能与 Master 上不同，因为如果语句顺序改变，则更新的行也会不同。

记录数据定义语言

数据定义语言（DDL）影响数据模式（schema）。例如 CREATE TABLE 和 ALTER TABLE 语句在文件系统中创建或改变对象，例如，表定义存储在 *.frm* 文件中，而数据库被表示为文件系统中的目录。因此服务器需要将这些信息保存在内部数据结构中。为了保护这个内部数据结构的更新，在修改表定义之前需要先获得锁。

只有一个锁用来保护这些数据结构，所以数据库对象的创建、更新和破坏都可能带来性能问题。例如创建和销毁临时表——临时表是通过创建中间结果集来执行计算任务的常用技术。

如果创建和销毁大量的临时表，通常可以通过减少创建临时表的数目（从而也减少临时表的销毁）来改善性能。

记录查询

对于基于语句的复制来说，最常见的 binlog 事件是 Query 事件，它用来存储 Master 上执行的语句。除了实际执行的语句外，该事件还包含执行语句必需的附加信息。

回想一下，二进制日志有很多用途，其中记录的语句一般与 Master 上执行的语句顺序不同。有时候可以向服务器回放二进制日志以执行 PITR，有时候可以从事件序列的中间位置开始复制（复制开始前 Slave 已经恢复了备份）。而且，数据库管理员（DBA）还可以手动调整二进制日志来解决问题。

所有这些情况下，事件都在不同的上下文（*context*）下执行。上下文是指服务器执行语句时必须知道的隐式（implicit）信息，以保证语句能够正确执行。例如：

- 当前数据库
 如果语句中引用了表、函数或过程，而没有指定哪个数据库，则默认使用当前数据库。
- 用户自定义变量的值
 如果语句中引用了用户自定义变量，则这个变量的值是隐式的。
- RAND 函数的种子
 RAND 函数是基于伪随机数的函数，即生成一系列可再生的数字，看上去是随机的，但实际上是均匀分布的。该函数并不是真正随机的，而是从一个种子数字开始，然后应用一个伪随机函数来产生确定性的数字序列。这意味着如果种子相同，RAND 函数总会返回相同的数字。所以，语句中的种子是隐式信息。
- 当前时间
 显然，语句开始执行的时间是隐式信息。但如果函数调用依赖于当前时间，那么保证时间的正确性就非常重要，因为如果语句分别在 Master 和 Slave 上的执行时间有延迟，则语句将返回不同的结果，例如 NOW 和 UNIX_TIMESTAMP 函数。
- AUTO_INCREMENT 字段的插入值
 向表中插入行，如果该行含有 AUTO_INCREMENT 类型的字段，那么新插入的行是隐式的，因为它与前面的行有关。
- 调用 LAST_INSERT_ID 的返回值
 如果语句调用了 LAST_INSERT_ID 函数，那么其值取决于前一个语句插入的值，因此也是隐式的。
- 线程 ID
 有些语句的线程 ID 是隐式信息。例如，如果某个语句使用了临时表或调用了 CURRENT_ID 函数，那么该语句的线程 ID 是隐式的。

无论在 Slave 还是 Master 上，宕机或重启后重放语句的时候，语句执行的上下文都是未知的，必须将隐式信息显式化，并写入二进制日志。根据隐式信息的种类不同，其实现

略有差异。

除了前面列出的隐式信息以外，还有些信息对触发器和存储程序的执行是隐式的，我们将单独在后面的“触发器、事件和存储例程”一节中讲述。

下面我们逐个考虑，描述每一种隐式信息存在的问题，以及服务器如何处理。

当前数据库

向 Query 事件添加一个特殊字段，记录当前数据库。处理 LOAD DATA INFILE 语句的事件中也有这个字段（详见后面的“LOAD DATA INFILE 语句”），所以这里的描述也适用于该语句。当前数据库还有一个重要用途，即数据库过滤，这将在本章后面介绍。

53

当前时间

有 5 个函数需要利用当前时间计算值：NOW，CURDATE，CURTIME，UNIX_TIMESTAMP 和 SYSDATE。前四个函数的返回值基于语句的开始执行时间，而 SYSDATE 将返回 time(2) 的值。间隔某段时间分别执行 NOW 和 SYSDATE，通过比较发现它们之间的区别。

```
mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();
+---------------------+----------+---------------------+
| SYSDATE()           | SLEEP(2) | SYSDATE()           |
+---------------------+----------+---------------------+
| 2010-03-27 22:27:36 |        0 | 2010-03-27 22:27:38 |
+---------------------+----------+---------------------+
1 row in set (2.00 sec)
mysql> SELECT NOW(), SLEEP(2), NOW();
+---------------------+----------+---------------------+
| NOW()               | SLEEP(2) | NOW()               |
+---------------------+----------+---------------------+
| 2010-03-27 22:27:49 |        0 | 2010-03-27 22:27:49 |
+---------------------+----------+---------------------+
1 row in set (2.00 sec)
```

这两个函数都会计算时间值，但是 NOW 返回的是语句的开始执行时间，而 SYSDATE 返回的是 time(2) 的时间。

为了正确处理这些时间函数，事件将存储一个时间戳，表明事件何时开始执行。然后将这个时间戳的值从事件复制到 Slave 执行进程，如果事件开始执行，则使用这个值来计算那些时间函数的值。

由于 SYSDATE 直接调用 time(2)，这对复制来说是不安全的，可能导致 Master 和 Slave 上执行的返回值不同，所以除非你真的想把实际的时间插入表中，否则最好慎用这个函数。

上下文事件

与语句有关的隐式信息需要满足以下条件：

- 如果语句包含对用户定义变量的引用（如例 3-1 中），就需要将用户定义变量的值写入二进制日志。
- 如果语句中包含 RAND 函数的调用，则需要将伪随机种子写入日志。
- 如果语句包含对 LAST_INSERT_ID 函数的调用，则需要将最后插入的 ID 写入二进制日志。
- 如果语句向含有 AUTO_INCREMENT 字段的表插入数据，则需要将这个字段（或多个字段）的值写入二进制日志。

54

例3-1：含有用户定义变量的语句

```
SET @value = 45;
INSERT INTO t1 VALUES (@value);
```

所有这些例子中，在写入包含查询的事件之前，需要向二进制日志中写入一个或多个上下文事件（*context events*）。由于查询（Query）事件之前可能有多个上下文事件，所以二进制日志可以一并处理多个用户定义变量和 RAND 函数，或（几乎）任何前面列出的情况的组合。二进制日志通过下面的事件来存储必需的上下文信息：

- User_var
 记录单个用户自定义的变量的变量名及其值。
- Rand
 记录 RAND 函数所用的随机数的种子。种子取自会话状态的内部。
- Intvar
 如果插入 AUTO_INCREMENT 字段，该事件记录在语句开始前，表内部的自动增量计数器的值。
 如果语句包含 LAST_INSERT_ID 函数调用，该事件记录这个函数在语句中的返回值。

例 3-2 给出了生成上下文事件的例子，以及 SHOW BINLOG EVENTS 命令的执行结果。注意每个语句之前可能有若干上下文事件。

例3-2：具有上下文事件的查询事件

```
master> CREATE TABLE t1 (a INT AUTO_INCREMENT PRIMARY KEY, b INT, c CHAR(64));
Query OK, 0 rows affected (0.00 sec)
master> SET @foo = 12;
Query OK, 0 rows affected (0.00 sec)
master> SET @bar = 'Smoothnoodlemaps';
Query OK, 0 rows affected (0.00 sec)
master> INSERT INTO t1(b,c) VALUES (@foo,@bar), (RAND(), 'random');
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2 Duplicates: 0 Warnings: 0
master> INSERT INTO t1(b) VALUES (LAST_INSERT_ID());
Query OK, 1 row affected (0.00 sec)
master> SHOW BINLOG EVENTS FROM 238\G
*************************** 1. row ***************************
Log_name: mysqld1-bin.000001
Pos: 238
55 Event_type: Query
Server_id: 1
End_log_pos: 306
Info: BEGIN
*************************** 2. row ***************************
Log_name: mysqld1-bin.000001
Pos: 306
Event_type: Intvar
Server_id: 1
End_log_pos: 334
Info: INSERT_ID=1
*************************** 3. row ***************************
Log_name: mysqld1-bin.000001
Pos: 334
Event_type: RAND
Server_id: 1
End_log_pos: 369
Info: rand_seed1=952494611,rand_seed2=949641547
*************************** 4. row ***************************
Log_name: mysqld1-bin.000001
Pos: 369
Event_type: User var
Server_id: 1
End_log_pos: 413
Info: @`foo`=12
*************************** 5. row ***************************
Log_name: mysqld1-bin.000001
Pos: 413
Event_type: User var
Server_id: 1
End_log_pos: 465
Info: @`bar`=_latin1 0x536D6F6F74686E6F6F6... COLLATE latin1_swedish_ci
*************************** 6. row ***************************
Log_name: mysqld1-bin.000001
Pos: 465
Event_type: Query
Server_id: 1
End_log_pos: 586
Info: use `test`; INSERT INTO t1(b,c) VALUES (@foo,@bar), (RAND(), ...
*************************** 7. row ***************************
Log_name: mysqld1-bin.000001
Pos: 586
```

```
Event_type: Xid
Server_id: 1
End_log_pos: 613
       Info: COMMIT /* xid=44 */
*************************** 8. row ***************************
Log_name: mysqld1-bin.000001
Pos: 613
Event_type: Query
Server_id: 1
End_log_pos: 681
Info: BEGIN
*************************** 9. row ***************************
Log_name: mysqld1-bin.000001
Pos: 681
Event_type: Intvar
Server_id: 1
End_log_pos: 709
Info: LAST_INSERT_ID=1
*************************** 10. row ***************************
Log_name: mysqld1-bin.000001
Pos: 709
Event_type: Intvar
Server_id: 1
End_log_pos: 737
Info: INSERT_ID=3
*************************** 11. row ***************************
Log_name: mysqld1-bin.000001
Pos: 737
Event_type: Query
Server_id: 1
End_log_pos: 843
Info: use `test`; INSERT INTO t1(b) VALUES (LAST_INSERT_ID())
*************************** 12. row ***************************
Log_name: mysqld1-bin.000001
Pos: 843
Event_type: Xid
Server_id: 1
End_log_pos: 870
Info: COMMIT /* xid=45 */
12 rows in set (0.00 sec)
```


线程ID

有时候二进制日志还需要处理某个语句的 MySQL 会话线程 ID。如果调用了某些依赖于线程 ID 的函数，例如 CONNECTION_ID，就需要线程 ID 信息。当然，在处理临时表时更加重要。

临时表是具体到每个线程的，即允许同时存在两个同名的临时文件，只要它们是在不同

的会话中定义的。临时表能够有效地提高性能，但是临时表与二进制日志之间的处理需要特殊对待。

在服务器内部，通过创建晦涩的名字存储表定义来处理临时表。临时表的名字由服务器的进程 ID（process ID）、创建表的线程 ID（thread ID）和一个线程特定计数器决定，用于区分同一线程的不同表实例。这种命名方式使得在不同线程中创建的表互相区别，但只有当前线程 ID 被存储在二进制日志中时，每个语句才能访问其相应的表。

与二进制日志处理当前的数据库的方法类似，线程 ID 也作为一个独立的字段存储在查询（Query）事件中，因此可用来计算线程特定数据，并正确地处理临时表。

57 当写入查询事件时，存储在事件中的线程 ID 是从服务器的 pseudo_thread_id 变量中读取的。这意味着它可以在执行语句前被设置，但前提是你有 SUPER 权限。此服务器变量拟被 mysqlbinlog 用来发出正确语句，而通常不会被使用。

对于包含调用 CONNECTION_ID 函数的语句，或者是使用或创建一个临时表，查询事件在二进制日志中被标记成“线程特定”。由于线程 ID 永远在查询事件中出现，这个标记并不是必需的，而主要是避免 mysqlbinlog 打印不必要的赋值 pseudo_thread_id 变量。

LOAD DATA INFILE语句

使用 LOAD DATA INFILE 语句很容易将文件中的数据快速读入表中。但遗憾的是，这依赖于曾经讨论过的上下文事件中没有提到的某一类上下文：文件需要从文件系统读取。

为处理 LOAD DATA INFILE，MySQL 服务器采用了一套特殊的事件来用二进制日志处理文件传输。很快你就会看到，除了解决了 LOAD DATA INFILE 的问题之外，这使得该语句成为一个非常方便的工具，这个工具可以从 Master 传输大量的数据到 Slave。要正确地传递和执行 LOAD DATA INFILE 语句，需要引入一些新的事件：

- Begin_load_query
 这个事件开始传输文件中的数据。
- Append_block
 如果这个文件超过连接的数据包大小所允许的最大值，那么跟随在 Begin_load_query 事件后面的一个或多个 Append_block 事件的序列包含着这个文件的剩余部分。
- Execute_load_query
 这个事件是 Query 事件的特殊变种，它包含了在 Master 上执行的 LOAD DATA INFILE 语句。

即使这个事件所含的语句含有在 Master 上使用的文件名，这个文件也将不会被 Slave 所

探寻。相反，前面的 Begin_load_query 和 Append_block 事件提供的内容将被使用。

对 Master 上执行的每个 LOAD DATA INFILE 语句而言，被读取的文件被映射到一个内部文件支持的缓冲区，并在接下来的处理流程中使用。此外，一个唯一的文件 ID 被分配给该执行语句，并用于指向该语句读取的文件。

当语句在执行时，该文件的内容被写入二进制日志，作为以 Begin_load_query 事件开头的事件序列，Begin_load_query 事件表示新文件的开始，且这个事件序列后面紧跟着零个或多个 Append_block 事件。每个写入二进制的事件都不会超过包大小所允许的最大值，这个最大值由 max-allowed-packet 选项指定。

当整个文件读取到表中后，通过写 Execute_load_query 事件到二进制日志来终止语句的执行。这个事件包含了执行语句和分配给该执行语句的文件 ID。请注意，这并非是用户写的原始语句，而是重新创建的。



如果你正在读旧的二进制日志，你或许可以发现 Load_log_event、Execute_log_even 和 Create_file_log_event。这些都是在 MySQL 5.0.3 之前的版本中用于复制 LOAD DATA INFILE 的事件，它们已经被上文所述的事件所取代。

例 3-3，通过成功地执行一个 LOAD DATA INFILE 展示了如何将这些事件写入二进制日志。在这个例子中，你可以看见 info 字段被分配的文件 ID 是 1，且看到它用于作为执行语句的一部分的所有事件中，你还可以看到这个语句所使用的文件 *foo.dat* 包含超过所允许的最大数据包大小 16384，因此它被分成三个事件。

例3-3：成功地执行LOAD DATA INFILE

```
master> SHOW BINLOG EVENTS IN 'master-bin.000042' FROM 269\G
*************************** 1. row ***************************
   Log_name: master-bin.000042
        Pos: 269
 Event_type: Begin_load_query
  Server_id: 1
End_log_pos: 16676
       Info: ;file_id=1;block_len=16384
*************************** 2. row ***************************
   Log_name: master-bin.000042
        Pos: 16676
 Event_type: Append_block
  Server_id: 1
End_log_pos: 33083
       Info: ;file_id=1;block_len=16384
*************************** 3. row ***************************
   Log_name: master-bin.000042
```

```
        Pos: 33083
 Event_type: Append_block
  Server_id: 1
End_log_pos: 33633
       Info: ;file_id=1;block_len=527
*************************** 4. row ***************************
   Log_name: master-bin.000042
        Pos: 33633
 Event_type: Execute_load_query
  Server_id: 1
End_log_pos: 33756
       Info: use `test`; LOAD DATA LOCAL INFILE 'foo.dat' INTO ... ;file_id=1
4 rows in set (0.00 sec)
```

59

二进制日志过滤器

二进制日志过滤器可以通过两个选项从二进制日志中过滤语句：`binlog-do-db` 和 `binlog-ignore-db`（可以使用 `binlog - *-db` 同时调用这两个选项）。当你想过滤只属于特定数据库的语句时使用 `binlog-do-db`，而当你想忽略某个特定数据库而复制其他数据库时则使用 `binlog-ignore-db`。

这些选项可以多次使用，所以要过滤 `one_db` 数据库和 `two_db` 数据库，你必须在 *my.cnf* 文件中指定这两个选项，例如：

```
[mysqld]
binlog-ignore-db=one_db
binlog-ignore-db=two_db
```

MySQL 过滤事件的方式会令不熟悉的用户相当吃惊，所以我们将解释过滤器是如何工作的，并为如何避免一些主要的头痛问题提供一些建议。

图 3-3 展示了 MySQL 如何确定语句是否被过滤。过滤是在语句级完成的（无论整个语句都被过滤或者整个语句都被写入二进制日志）且 `binlog-*-db` 使用当前数据库以决定是否应该过滤该语句，而不是由语句所影响的表所在的数据库决定的。

为有助于理解这个行为，例 3-4 展示了更改不同数据库中的表的语句。每一行以 `test` 数据库作为当前数据库：

- 第一行改变了当前数据库 `test` 中的一张表，因为它不符合表名加数据库名。
- 第二行改变了不同于当前数据库的数据库中的表。
- 第三行改变了两个不同数据库中的两个表，这两个数据库都不是当前数据库。

例3-4：使用不同数据库的语句

```
USE bad; INSERT INTO t1 VALUES (1),(2);
```

```
USE bad; INSERT INTO good.t2 VALUES (1),(2);
USE bad; UPDATE good.t1, ugly.t2 SET a = b;
```

如果使用 `binlog-ignore-db=bad` 筛选 bad 数据库，考虑会发生什么。例 3-4 中的三个语句一个也不会被写入二进制日志，尽管第二个和第三个语句改变的是 good 和 ugly 数据库中的表。这初看起来可能有点奇怪——为什么不过滤基于改变的表所在的数据库的语句？考虑一下如果 *ugly* 数据库被筛选而不是 *bad* 数据库被筛选，第三个语句会发生什么。现在 UPDATE 中的一个数据库被筛除了而另一个没有，这将使服务器进入 catch-22 状况，所以只需过滤当前数据库就可以解决这个问题，而且这个规则适用于所有语句（除少数例外）。 60

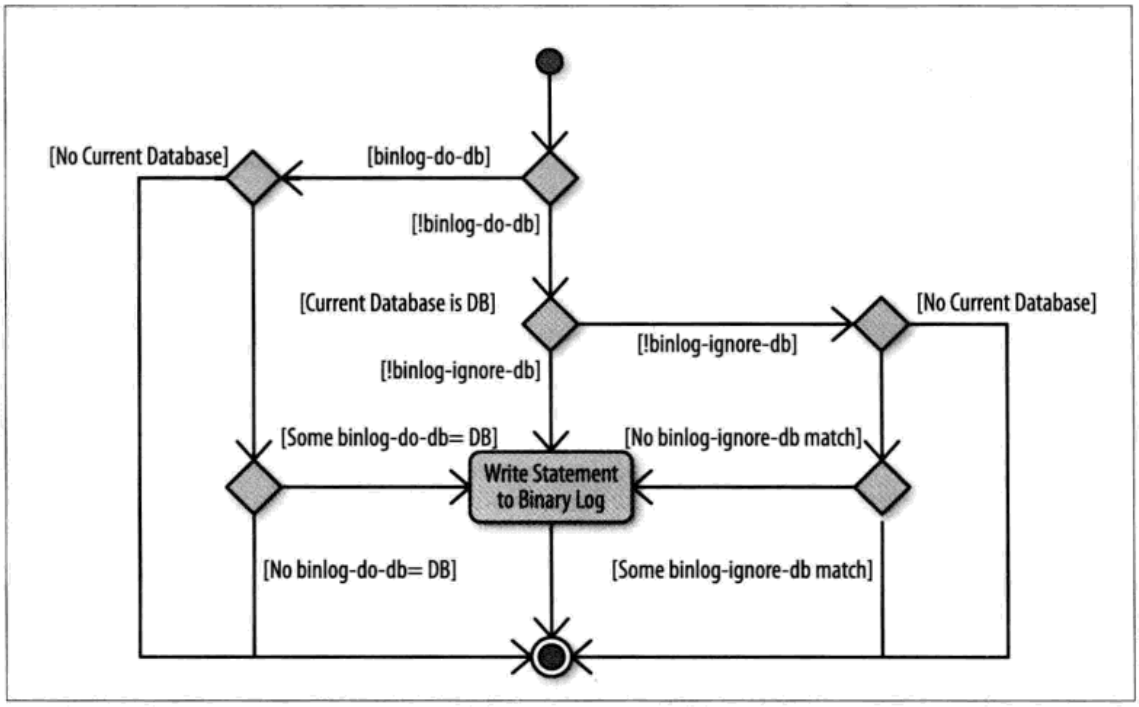


图3-3：*binlog-*-db*过滤器的逻辑



为避免在执行可能被过滤的语句时发生错误，请不要编写那些表名、函数名或存储过程名加上数据库名的语句。每当你要访问不同的数据库时，发出一个 USE 语句来使该数据库成为当前数据库。比如，使用

```
INSERT INTO other. book VALUES ('MySQL', 'Paul DuBois');
```

替换

```
USE other; INSERT INTO book VALUES ('MySQL', 'Paul DuBois');
```

此行为不适用于基于行的复制，我们将在第 6 章讨论基于行的复制。但由于基于行的复制可以在每个独立行改变的时候工作，因此它能够过滤行所在的实际表，而不是用当前数据库。

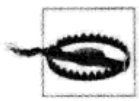
所以，当 binlog-do-db 和 binlog-ignore-db 同时被使用的时候会发生什么呢？例如，考虑一个配置文件包含了下面两个规则：

61
```
[mysqld]
binlog-do-db=good
binlog-ignore-db=bad
```

在这种情况下，下面的语句会不会被过滤呢？

```
USE ugly; INSERT INTO t1 VALUES (1);
```

根据图 3-3 中的图解，你可以看到只要有一个 binlog-do-db 规则存在，所有的 binlog-ignore-db 规则将完全被忽略，而且由于只包含了 *good* 数据库，所以上面的这个语句将被过滤。



由于评估 binlog-*-db 规则的方式，同时使用 binlog-do-db 和 binlog-ignore-db 规则是毫无意义的。不推荐使用 binlog-*-db 选项，因为二进制日志可用于复制和恢复。如果你从二进制日志过滤掉语句，在事件崩溃时，将无法从二进制日志恢复数据库。

触发器、事件和存储程序

记录日志时需要特别处理的一些其他结构是 *stored programs*，即触发器、事件和存储程序（最后一个是存储过程和存储函数的总称）。这些 *stored programs* 的有关二进制日志的处理包含一些共同的元素，所以将在本节一起讨论。下面主要介绍两种类型的处理语句：定义或销毁存储程序的语句和调用它们的语句。

定义或销毁储存程序的语句

在下面的例子中讨论了触发器，但同样的原则也适用于事件和存储程序的定义。要理解为什么服务器需要处理这些功能，特别是当把它们写入到二进制日志时，请考虑例 3-5 中的代码。

在这个例子中，*employee* 表存储了设想的系统中的所有雇员信息，*log* 表保留了有趣信息的日志。请注意，*log* 表有一个 timestamp 字段用于记录每个更改的时间，而 *employee* 表的主键是 name 字段。还有一个 status 字段用来表明增加的人或事务是成功还是失败。

为了跟踪员工信息变化的信息（例如出于审计目的）创建了三个触发器，以便当一名员工被添加、删除或更改时，更新的日志条目会被添加到日志表。

请注意这些触发器都是after触发器，这意味着只有当语句执行成功时条目才会被添加。失败的语句将不会被记录。我们稍后将扩展这个例子，以使失败的尝试也会被记录。

例3-5：定义员工管理的表和触发器 62

```
CREATE TABLE employee (
   name CHAR(64) NOT NULL,
   email CHAR(64),
   password CHAR(64),
   PRIMARY KEY (name)
);

CREATE TABLE log (
   id INT AUTO_INCREMENT,
   email CHAR(64),
   message TEXT,
   ts TIMESTAMP,
   PRIMARY KEY (id)
);

CREATE TRIGGER tr_employee_insert_after AFTER INSERT ON employee FOR EACH ROW
   INSERT INTO log(email, status, message)
     VALUES (NEW.email, 'OK', CONCAT('Adding employee ', NEW.name));

CREATE TRIGGER tr_employee_delete_after AFTER DELETE ON employee FOR EACH ROW
   INSERT INTO log(email, status, message)
     VALUES (OLD.email, 'OK', 'Removing employee');

delimiter $$
CREATE TRIGGER tr_employee_update_after AFTER UPDATE ON employee FOR EACH ROW
BEGIN
  IF OLD.name != NEW.name THEN
    INSERT INTO log(email, status, message)
      VALUES (OLD.email, 'OK',
              CONCAT('Name change from ', OLD.name, ' to ', NEW.name));
  END IF;
  IF OLD.password != NEW.password THEN
    INSERT INTO log(email, status, message)
      VALUES (OLD.email, 'OK', 'Password change');
  END IF;
  IF OLD.email != NEW.email THEN
    INSERT INTO log(email, status, message)
      VALUES (OLD.email, 'OK', CONCAT('E-mail change to ', NEW.email));
  END IF;
END $$
delimiter ;
```

有了这些触发器的定义，现在可以按例 3-6 所示添加和删除雇员。正如你所看到的，一个雇员被添加、修改和删除，每个操作都被记录到 *log* 表。

添加、删除和修改雇员的操作可以由有权访问 *employee* 表的用户完成，但是谁访问 *log* 表呢？在这个例子中，可以操纵雇员表的用户能够更改 *log* 表。这有许多原因，但它们归根结底还是出于维护、审计、合法权威披露等目的，而信任 *log* 表的内容。因此，DBA(数据库管理员) 可以选择让许多用户访问 *Employee* 表，而同时非常严格地控制对 *log* 表的访问。

为确保触发器可以成功地在一个被高度保护的表上执行，它们被定义这个触发器的用户而不是更改 *employee* 表内容的用户来执行。因此，例 3-5 的 CREATE TRIGGER 语句由具备添加 *log* 表的权限的 DBA 执行，而例 3-6 中修改雇员信息是通过具有修改 *employee* 表权限的用户来执行的。

当例 3-6 中的语句被执行时，员工管理账户用于更新 *employee* 表中的条目，而 DBA 权限则被用于增加 *log* 表。员工管理账户不能够用于添加或删除 *log* 表的条目。

顺便说一句，例 3-6 在语句中使用用户变量前，将密码分配给用户变量。这样做是为了避免在纯文本中发送敏感数据到另一台服务器；更多细节可在后面的“安全与二进制日志”中找到。

例3-6：添加、删除和修改用户

```
master> SET @pass = PASSWORD('xyzzy');
Query OK, 0 rows affected (0.00 sec)

master> INSERT INTO employee VALUES ('mats', 'mats@example.com', @pass);
Query OK, 1 row affected (0.00 sec)

master> UPDATE employee SET name = 'matz' WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

master> SET @pass = PASSWORD('foobar');
Query OK, 0 rows affected (0.00 sec)

master> UPDATE employee SET password = @pass WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

master> DELETE FROM employee WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.00 sec)

master> SELECT * FROM log;
+----+-----------------+------------------------------+--------------------+
```

```
| id | email            | message                        | ts                  |
+----+------------------+--------------------------------+---------------------+
|  1 | mats@example.com | Adding employee mats           | 2010-01-13 18:56:08 |
|  2 | mats@example.com | Name change from mats to matz  | 2010-01-13 18:56:11 |
|  3 | mats@example.com | Removing employee              | 2010-01-13 18:57:11 |
+----+------------------+--------------------------------+---------------------+
3 rows in set (0.00 sec)
```

安全和二进制日志

一般来说，一个有 REPLICATION SLAVE 权限的用户拥有读取 Master 上发生的所有事件的权限，因此为了确保安全应使该账户不被损害。详情不在本书讨论范围之内，但是这里介绍一些预防措施的例子：

- 尽可能使从防火墙外无法登录该账户。
- 记录所有试图登录到该账户的日志，并将日志放置在一个单独的安全服务器上。
- 加密 Master 和 Slave 间所用的连接，例如 MySQL 的 built-in SSL (Secure Sockets Layer) 支持。

即使这个账户已经安全了，还存在一些没必要放在二进制日志中的信息，因此首先不存储在那里也是有道理的。

较为常见的一个敏感信息就是密码。当执行改变服务器上的表的语句，并且它包含访问这个表所必需的密码的时候，包含密码的事件会被写入二进制日志。

一个典型的例子是：

```
UPDATE employee SET pass = PASSWORD('foobar')
WHERE email = 'mats@example.com';
```

如果复制是正确的，最好重写这个没有密码的语句。可以通过以下方法实现：计算和存储哈希密码到用户定义变量，然后在表达式中使用它：

```
SET @password = PASSWORD('foobar');
UPDATE employee SET pass = @password WHERE email = 'mats@example.com';
```

由于 SET 语句没有被复制，原来的密码将不会存储在二进制日志中，而仅在执行该语句的时候存储在服务器的内存中。

只要存储 *password hash* 到表中，而不需要纯文本密码，这种方法行之有效。如果原始密码是直接存储在表中的，就有没有办法阻止密码在二进制日志中结束。但存储哈希密码在任何情况下都是一个标准的好做法，可以防止有人通过学习密码将原始数据弄到手。

> 封装连接为加密 Master 和 Slave 之间的连接提供了一些保护，但如果二进制日志本身被攻破，加密连接也无能为力。

回忆一下前面讨论的隐式信息，你可能已经注意到用户执行一行代码和定义一个触发器都是隐式的。正如你将在第 6 章看到的，无论是触发器的定义者或是调用者，在 Slave 上执行触发器都是危险的，当在 Slave 上执行该语句时，用户信息实际上都被忽略了。无论如何，当在服务器上回放二进制日志时，这些信息都是重要的，例如执行 PITR 时。

65 为在服务器上重播二进制日志，毫无问题地处理各种表的权限，有必要用具备 SUPER 权限的用户执行所有语句。但触发器没有被定义使用 SUPER 权限，所以重要的是以正确的用户作为触发器的定义者去重新创建触发器。如果以 SUPER 权限定义了一个触发器，而不是由最初定义触发器的用户，可能会导致权限的放大。

为允许 DBA 指定用户执行触发器，CREATE TRIGGER 语法包括一个可选的 DEFINER 从句。如果没有给语句指定 DEFINER（如例 3-7 的情况），该语句添加 DEFINER 从句后被重写到二进制日志中，且将使用当前用户作为定义者。这意味着 INSERT 触发器的定义将出现在二进制日志中，如例 3-7 所示，它列出了创建触发器的定义者的账户（root@localhost），这就是我们在这种情况下想要的。

例3-7：在二进制日志中的一个CREATE TRIGGER语句

```
master> SHOW BINLOG EVENTS FROM 92236 LIMIT 1\G
*************************** 1. row ***************************
   Log_name: master-bin.000038
        Pos: 92236
  Event_type: Query
   Server_id: 1
End_log_pos: 92491
       Info: use `test`; CREATE DEFINER=`root`@`localhost` TRIGGER ...
1 row in set (0.00 sec)
```

调用触发器和存储程序的语句

从定义到调用，我们可以询问 Master 的触发器在复制过程中是如何处理的。其实它们什么都没有处理。

调用触发器的语句被记录到二进制日志，但它没有连接到特定的触发器。相反，当 Slave 执行该语句时，它会自动执行受该语句影响的表相关联的所有触发器。这意味着可以在 Master 和 Slave 上有不同的触发器。Master 上的触发器将在 Master 上被调用，同时 Slave 上的触发器将在 Slave 上被调用。例如，如果添加 log 表条目的触发器在 Slave 上不是必需的，通过从 Slave 上消除触发器可以提高性能。

然而，在语句援用触发器前，正确复制所必需的一切上下文事件都将写入二进制日志，即使只是触发器中的语句需要的上下文事件。因此，例 3-8 展示了在执行例 3-5 的 INSERT 语句后的二进制日志。请注意第一个事件记录了 *log* 表主键的 INSERT ID。这反映了 *log* 表在触发器中用法，但它似乎是多余的，因为 Slave 不会使用该触发器。

无论如何必须注意在 Master 和 Slave 上使用不同的触发器（或者根本没有触发器在 Master 或 Slave 上）是个例外，而当 Master 和 Slave 上都有触发器时，INSERT ID 是正确复制 INSERT 语句所必需的。

例3-8：在执行INSERT之后二进制日志的内容

```
master> SHOW BINLOG EVENTS FROM 93340\G
*************************** 1. row ***************************
   Log_name: master-bin.000038
        Pos: 93340
  Event_type: Intvar
   Server_id: 1
End_log_pos: 93368
       Info: INSERT_ID=1
*************************** 2. row ***************************
   Log_name: master-bin.000038
        Pos: 93368
  Event_type: User var
   Server_id: 1
End_log_pos: 93396h
       Info: @`pass`=_latin1
0x2A394235303033343342433532453239313131373245423532 41... COLLATE
latin1_swedish_ci
*************************** 3. row ***************************
   Log_name: master-bin.000038
        Pos: 93396
  Event_type: Query
   Server_id: 1
End_log_pos: 93537
       Info: use `test`; INSERT INTO employee VALUES ...
3 rows in set (0.00 sec)
```

存储过程

存储函数、存储过程和事件是已知的存储程序的通用名称。由于服务器对存储过程和存储函数的处理是截然不同的，存储过程将在本节描述，而存储函数将在下一节介绍。

存储程序的情况跟触发器在某些方面很类似，但在其他方面是不同的。像触发器一样，存储程序定义者提供了一个 DEFINER 从句，无论语句是否包含了它，它都必须明确添加到二进制日志中。但是存储程序的调用方式与触发器不同。

首先，让我们扩展例 3-6，这个例子中定义了员工和日志的表，使用一些实用工具来进行员工管理。虽然这些可以用标准的 INSERT，DELETE 和 UPDATE 语句来处理，但我们将使用存储过程来处理，以表明它们被写入二进制日志中涉及的一些问题。出于这些目的，让我们扩展例 3-9 中的函数，并用这些函数添加和删除员工。

67 例3-9：为管理员工定义的存储过程

```
delimiter $$
CREATE PROCEDURE employee_add(p_name CHAR(64), p_email CHAR(64),
                              p_password CHAR(64))
   MODIFIES SQL DATA
BEGIN
   DECLARE pass CHAR(64);
   SET pass = PASSWORD(p_pass)
   INSERT INTO employee(name, email, password)
     VALUES (p_name, p_email, pass);
END $$

CREATE PROCEDURE employee_passwd(p_email CHAR(64), p_password CHAR(64))
   MODIFIES SQL DATA
BEGIN
  DECLARE pass CHAR(64);
  SET pass = PASSWORD(p_password)
  UPDATE employee SET password = pass WHERE email = p_email;
END $$

CREATE PROCEDURE employee_del(p_name CHAR(64))
   MODIFIES SQL DATA
BEGIN
  DELETE FROM employee WHERE name = p_name;
END $$
delimiter ;
```

对于 employee_add 和 employee_passwd 程序，出于前面已经解释过的理由，我们已经将加密密码提取到一个单独的变量中，但是 employee_del 程序因为没有别的需要而只包含一个 DELETE 语句。binlog 条目对应的函数是：

```
master> SHOW BINLOG EVENTS FROM 97911 LIMIT 1\G
*************************** 1. row ***************************
   Log_name: master-bin.000038
        Pos: 97911
 Event_type: Query
  Server_id: 1
End_log_pos: 98275
       Info: use `test`; CREATE DEFINER=`root`@`localhost`PROCEDURE ...
1 row in set (0.00 sec)
```

正如预期的那样，这个程序的定义在写入二进制日志之前被加上了 DEFINER 从句，但除

此之外，该程序的主体还是原封不动的。请注意，CREATE PROCEDURE 语句被作为一个查询事件来复制，因为全都是 DDL 语句。

在这方面，二进制日志对存储程序与触发器的处理方式是类似的。但在调用上与触发器有显著的区别。例 3-10 调用这个程序去添加员工，并显示了二进制日志的生成内容。

例3-10：调用存储过程


```
master> CALL employee_add('chuck', 'chuck@example.com', 'abrakadabra');
Query OK, 1 row affected (0.00 sec)
master> SHOW BINLOG EVENTS FROM 104033\G
*************************** 1. row ***************************
   Log_name: master-bin.000038
        Pos: 104033
 Event_type: Intvar
  Server_id: 1
End_log_pos: 104061
       Info: INSERT_ID=1
*************************** 2. row ***************************
   Log_name: master-bin.000038
        Pos: 104061
 Event_type: Query
  Server_id: 1
End_log_pos: 104416
       Info: use `test`; INSERT INTO employee(name, email, password)
             VALUES ( NAME_CONST('p_name',_latin1'chuck' COLLATE 'latin1_swedish_ci'),
             NAME_CONST('p_email',_latin1'chuck@example.com' COLLATE 'latin1_
swedish_ci'),
             NAME_CONST('pass',_latin1'*FEB349C4FDAA307A...' COLLATE 'latin1_
swedish_ci'))
2 rows in set (0.00 sec)
```

在例 3-10 中，有四件事需要注意：

- CALL 语句没有被写入二进制日志。取而代之的是，执行的语句作为调用的结果被写入二进制日志。换句话说，该存储过程的主体被展开后放入二进制日志中。
- 该语句改写为不包含任何对存储过程的参数的引用，也就是说，p_name，p_email 和 p_password。取而代之的是，使用 NAME_CONST 函数为每个参数创建一个单值的结果集。
- 局部声明的变量 pass 也被换成了 NAME_CONST 表达式，其中第二个参数包含加密的密码。
- 正如当调用触发器的语句被写入二进制日志，调用存储过程的语句以 Intvar 事件作为前缀，这个 Intvar 事件包含了在添加员工加入到 *log* 表时使用的 insert ID。

由于在存储程序之外，无论是参数名称还是局部声明的名称都是不可用的，NAME_CONST

SELECT怎么样呢？通常，SELECT语句不会被写入二进制日志，因为它们不会改变任何数据，但是一个含有存储函数的SELECT语句是个例外。当执行存储函数，服务器注意到添加一行到 *log* 表而且标记该语句作为一个“updating”语句，这意味着它会被写入二进制日志。

存储函数和权限

CREATE ROUTINE权限是定义一个存储过程或存储函数所必需的。严格说创建一个存储程序不需要其他权限，但由于它通常根据定义者的权限执行，所以定义一个存储程序并不在意程序的定义者是否有权限去读写存储过程引用的表。

然而Slave上的复制线程在不进行权限检查的情况下执行。这留下了严重的安全漏洞，允许所有具有CREATE ROUTINE权限的用户提升其权限，且可以在Slave上执行任何语句。

在5.0之前的MySQL版本中，这并不会导致问题，因为当在Master上执行时语句的所有路径都被勘察过。在Master上违规的权限将阻止语句写入二进制日志，因此用户在Slave上就不能访问Master上禁止的对象。然而，随着存储程序的引入，它有可能创造条件执行路径，这样服务器在执行存储程序时不会勘察所有的路径。

由于存储过程被展开了，在Master上执行的具体语句也将会在Slave上执行，而且由于只有在Master上成功执行的语句才会被记录，因此它不可能访问其他对象，这一点与存储函数不同。

如果一个存储函数被定义含有SQL SECURITY INVOKER，恶意用户可以精心设计一个函数，使其在Master和Slave上的执行不同。该安全漏洞可以被安葬在Slave上执行的分支。如下面的例子所示：

```
CREATE FUNCTION magic()
  RETURNS CHAR(64)
  SQL SECURITY INVOKER
BEGIN
  DECLARE result CHAR(64);
  IF @@server_id <> 1 THEN
     SELECT what INTO result FROM secret.agents LIMIT 1;
     RETURN result;
  ELSE
     RETURN 'I am magic!';
  END IF;
END $$
```

在 Master 上执行一段代码（ELSE 分支），而在 Slave 上执行单独的一段代码（IF 分支）时权限检查是被禁用的。其结果是提升用户的权限从 CREATE ROUTINE 到相当于 SUPER。

71 请注意，如果该函数被定义含有 SQL SECURITY DEFINER 时，这个问题不会出现，因为该函数会以该用户的权限去执行，而且将在 Slave 上被封锁。

为预防权限在 Slave 上被扩大，MySQL 默认要求 SUPER 权限来定义存储函数。但是由于存储函数是非常有用的，而且一些数据库管理员信任他的用户能创建恰当的函数，因此这个检查可以通过 log-bin-trust-function-creators 选项来禁止。

Events

events 功能是 MySQL 扩展的，并不是标准 SQL 的一部分。Events 不能跟 binlog 事件相混淆，它由存储程序处理，通过一个特殊的事件调度器来定期执行。

与其他存储程序类似，事件的定义和 DEFINER 从句一起被记录到日志中。由于事件由事件调度器调用，因此它们总是以定义者执行而且不会存在存储函数的安全漏洞。当事件被执行时，该语句被直接写入二进制日志。

由于事件将在 Master 上执行，它们在 Slave 上是自动被禁止的，因此将不会在那里执行。如果事件不是被禁止的，它们将在 Slave 上执行两次：第一次是因为 Master 执行该事件然后复制这个更改到 Slave，还有一次是 Slave 直接执行这个事件。



由于这个事件在 Slave 上被禁止，所以当 Slave 出于某些原因丢失了 Master 时，有必要允许在 Slave 上执行这些事件。

因此，例如，像第 4 章中描述的那样升级一个 Slave，不要忘记打开允许从 Master 上复制过来的事件。利用下面的语句很容易做到：

```
UPDATE mysql.events
   SET Status = ENABLED
 WHERE Status = SLAVESIDE_DISABLED;
```

这个检查的目的是只允许打开从 Master 上复制过来的被禁止事件，而那里可能存在一些因其他原因而禁止的事件。

特殊结构

尽管基于语句的复制通常是简单的，但一些特殊结构必须小心处理。回想一下，为了在

Slave 上正确执行语句，语句的上下文一定要正确。尽管前面讨论的上下文事件处理了上下文的一部分，但是一些有额外上下文的结构没有作为复制过程的一部分被传递。

LOAD_FILE函数

72

LOAD_FILE 函数让你可以获取一个文件，然后使用它作为表达式的一部分。虽然有时候很方便，但这个文件必须正确地存在于要复制的 Slave 上。因为在复制的过程中，作为 LOAD DATA INFILE 的文件是不会被传输的。例如，采用下面的语句插入文档到一个表中：

```
Master-> INSERT INTO document(author, body)
      ->   VALUES ('Mats Kindahl', LOAD_FILE('go_intro.xml'));
```

可以用 LOAD DATA INFILE 来重写上面的语句。在这个例子中，你要小心文档中不能存在以特殊字符作为字段和行的分隔符，因为你将以一个单独的列来读取整个文件的内容。

```
master> LOAD DATA INFILE 'go_intro.xml' INTO TABLE document
    ->    FIELDS TERMINATED BY '@*@' LINES TERMINATED BY '&%&'
    ->    (author, body) SET author = 'Mats Kindahl';
```

一个可选的方案是在一个用户定义变量中存储文件内容，然后在语句中使用它。

```
master> SET @document = LOAD_FILE('go_intro.xml');
master> INSERT INTO document(author, body) VALUES('Mats Kindahl, @document);
```

非事务性的变化和错误处理

到目前为止，我们只考虑事务的变化，完全没有查看错误处理。对于事务性变化，错误处理是非常简单的：一个语句试图改变事务的表而失败了，将不会对这个表有任何影响。这就是具有一个事务性的系统的最重要之处——因此该语句试图引起的改变可以被忽略。这同样适用于回滚的事务：它们没有对表产生任何影响，因此仅被丢弃掉，而不会引起 Master 和 Slave 之间不一致的风险。

73

MySQL 的一个特长是提供非事务性的存储引擎。这可以提供一些速度上的优势，因为该存储引擎不需要管理事务引擎所使用的事务日志，而且它给予了一些磁盘访问上的优化。从复制的角度看，无论如何，非事务性引擎需要特殊考虑。

最重要、需要注意的方面是复制不能处理任意的非事务性引擎，但要做出一些关于它们如何工作的假设。随着在 5.1 版本中引入基于行的复制，这些限制中的一些被取消了（将在第 6 章中介绍）但是即使在那样的情况下，也不能处理任意的存储引擎。

从复制的角度看，使这个问题更加复杂的一个特性是，这可以在同一事务中甚至是在同一个语句中混合事务性和非事务性引擎。

继续前面使用的例子，考虑到例 3-13，例 3-5 中的 *log* 表给定的是非事务性存储引擎，而 *employee* 表则给定的是事务性存储引擎。我们使用非事务性 MyISAM 存储引擎来提高 *log* 表的速度，而为 *employee* 表保持事务性行为。

可以进一步扩展这个例子，通过创建一对 insert 触发器来追踪增加新员工的失败尝试：一个 before 触发器和一个 after 触发器。如果一个管理员看到日志中的一个条目的 status 字段为 FAIL，这意味着 before 触发器运行了，但是 after 触发器没有运行，因此一个增加新员工的尝试失败了。

例3-13：定义带有存储引擎的log和employee表

```
CREATE TABLE employee (
   name CHAR(64) NOT NULL,
   email CHAR(64),
   password CHAR(64),
   PRIMARY KEY (email)
) ENGINE = InnoDB;

CREATE TABLE log (
   id INT AUTO_INCREMENT,
   email CHAR(64),
   message TEXT,
   status ENUM('FAIL', 'OK') DEFAULT 'FAIL',
   ts TIMESTAMP,
   PRIMARY KEY (id)
) ENGINE = MyISAM;

delimiter $$
CREATE TRIGGER tr_employee_insert_before BEFORE INSERT ON employee FOR EACH ROW
BEGIN
  INSERT INTO log(email, message)
    VALUES (NEW.email, CONCAT('Adding employee ', NEW.name));
  SET @LAST_INSERT_ID = LAST_INSERT_ID();
END $$
delimiter ;

CREATE TRIGGER tr_employee_insert_after AFTER INSERT ON employee FOR EACH ROW
   UPDATE log SET status = 'OK' WHERE id = @LAST_INSERT_ID;
```

这个改变对二进制日志有什么影响？

首先，让我们考虑一下例 3-6 的 INSERT 语句。假设这个语句不是在一个事务中，且 AUTOCOMMIT 设为 1，这个语句本身就将成为一个事务。如果语句无错地执行，每件事都将按预期进行，而且该语句将作为一个 Query 事件写入二进制日志。

74

现在，考虑下如果重复执行同一个员工的 INSERT 语句会发生什么。由于 *email* 字段是主

关键字，当尝试插入的时候将产生重复键的错误，但是这个语句会发生什么呢？会不会写入二进制日志呢？

让我们来看一看：

```
master> SET @pass = PASSWORD('xyzzy');
Query OK, 0 rows affected (0.00 sec)

master> INSERT INTO employee(name,email,pass)
    ->   VALUES ('mats','mats@example.com',@pass);
ERROR 1062 (23000): Duplicate entry 'mats@example.com' for key 'PRIMARY'
master> SELECT * FROM employee;
+------+-----------------+-------------------------------------------+
| name | email           | password                                  |
+------+-----------------+-------------------------------------------+
| mats | mats@example.com | *151AF6B8C3A6AA09CFCCBD34601F2D309ED54888 |
+------+-----------------+-------------------------------------------+
1 row in set (0.00 sec)
master> SHOW BINLOG EVENTS FROM 38493\G
*************************** 1. row ***************************
   Log_name: master-bin.000038
        Pos: 38493
 Event_type: User var
  Server_id: 1
End_log_pos: 38571
       Info: @`pass`=_latin1 0x2A3135314146364238433341364141303946434...
             COLLATE latin1_swedish_ci
*************************** 2. row ***************************
   Log_name: master-bin.000038
        Pos: 38571
 Event_type: Query
  Server_id: 1
End_log_pos: 38689
       Info: use `test`; INSERT INTO employee(name,email,pass) VALUES ...
2 rows in set (0.00 sec)
```

正如你所看到的那样，这个语句被写入二进制日志，即使 *employee* 表是事务性的而语句执行失败了。注意看通过 SELECT 显示的表的内容，只有单个 employee，这证明这个语句已经被回滚了。但为什么这个语句还会写入二进制日志呢？

查看 log 表将发现原因。

```
master> SELECT * FROM log;
+----+-----------------+---------------------+-----+---------------------+
| id | email | message  | status | ts |
+----+-----------------+---------------------+-----+---------------------+
|  1 | mats@example.com | Adding employee mats | OK  | 2010-01-13 15:50:45 |
|  2 | mats@example.com | Name change from ... | OK  | 2010-01-13 15:50:48 |
```

```
|  3 | mats@example.com | Password change      | OK  | 2010-01-13 15:50:50 |
|  4 | mats@example.com | Removing employee    | OK  | 2010-01-13 15:50:52 |
|  5 | mats@example.com | Adding employee mats | OK  | 2010-01-13 16:11:45 |
|  6 | mats@example.com | Adding employee mats | FAIL| 2010-01-13 16:12:00 |
+----+------------------+----------------------+-----+---------------------+
6 rows in set (0.00 sec)
```

看最后一行，状态是 FAIL。这一行是由 before 触发器 tr_employee_insert_before 插入到表中的。由于二进制日志忠诚地代表 Master 上数据库的改变，如果在语句中或在作为执行语句的结果而被执行的触发器中存在一些非事务性的更改，就有必要将该语句写入二进制日志。由于这个语句失败了，after 触发器 tr_employee_insert_after 就没有被执行，因此状态就仍是 before 触发器执行时写的 FAIL。

由于在 Master 上这个语句执行失败了，因此失败的信息同样也要写入二进制日志。MySQL 服务器是通过 Query 事件中一个错误代码字段来登记导致该语句失败的确切错误代码来处理的，这个字段接着跟事件一起被写入二进制日志。

当使用 SHOW BINLOG EVENTS 命令时，这个错误代码是看不见的，但是可以通过 mysqlbinlog 工具来查看它，这个工具将在下一章介绍。

记录事务

现在可以看单独的语句和上下文信息是如何被一起写入二进制日志的。事务需要额外的处理。

一个事务可以在一些不同的情况下开始：

- 当用户执行 START TRANSACTION 或 BEGIN 时。
- 当 AUTOCOMMIT=1，且访问事务性表的语句开始执行时。请注意，只写入非事务性的表的语句（例如只写入 MyISAM 表）不启动事务。
- 当 AUTOCOMMIT=0 且先前的事务默认已经被提交或终止了（通过执行一个默认提交的语句）或默认使用 COMMIT 或 ROLLBACK。

并不是在事务开始后执行的每个语句都是事务的一部分。需要从二进制日志仔细考虑例外情况。

76 非事务性语句就其本身的定义而言，不是事务的一部分。当它们被执行时，它们立即生效而不是等事务来提交。这也意味着没必要回滚它们。它们不影响活动的事务：在非事务性语句后执行的任何事务性的语句都将被添加到当前活动的事务中。

此外，一些语句默认提交。基于其隐式提交的原因，它们可以分成三组：

- 写文件的语句

 大多数 DDL 语句（CREATE，ALTER，等等）（有一些例外）在开始执行前对任何未解决的事务做一个隐式提交，且在它们结束后再做一个隐式提交。这些语句修改文件系统中的文件，因此不是事务性的。
- 修改 *mysql* 数据库中表的语句

 创建、删除或修改用户账户或用户权限的所有语句都是隐式提交的，而且不是事务的一部分。从内部而言，这些修改 *mysql* 数据库中的表的语句都是非事务性的。

 在 5.1.3 之前的 MySQL 版本中，这些语句都不会导致隐式提交，但由于被写入非事务性表中，它们被看做是非事务性语句。正如你将马上看到的那样，这导致了一些矛盾，因此为了这些语句能跨越以上的几个版本，隐式提交被加进来。
- 为实际理由而需要隐式提交的语句

 在很多情况下，锁表的语句、用于管理的语句和 LOAD DATA INFILE 会导致隐式提交，因为执行需要使它们正常工作。

 引起隐式提交的语句明显不是任何事务的一部分，因为一切活动的事务都在执行开始前提交。你可以在在线的 MySQL 参考手册中找到引起隐式提交的语句的完整列表。

事务缓存

二进制日志可以有与它们实际执行顺序不同排序的语句，因为它联合了每个事务中的所有语句来使得它们在一起。多个会话可以在服务器上同步执行事务，事务存储引擎维护它们自己的事务日志来确保每个事务正确执行。这些日志对用户来说是不可见的。相反，二进制日志按照其提交的顺序展示了来自所有会话中的一切事务，仿佛每个都按照顺序执行。

为确保每个事务都作为一个单元被写入二进制日志，服务器需要将在不同线程中执行的语句分开。每当提交一个事务时，服务器将所有语句作为事务的一部分，当做单个单元写入二进制日志。出于这个原因，如图 3-4 所示，服务器为每个线程保持一个事务缓存。事务中执行的每个语句都被放在事务缓存中，然后事务缓存的内容被复制到二进制日志中，而且在事务提交时被清空。

77

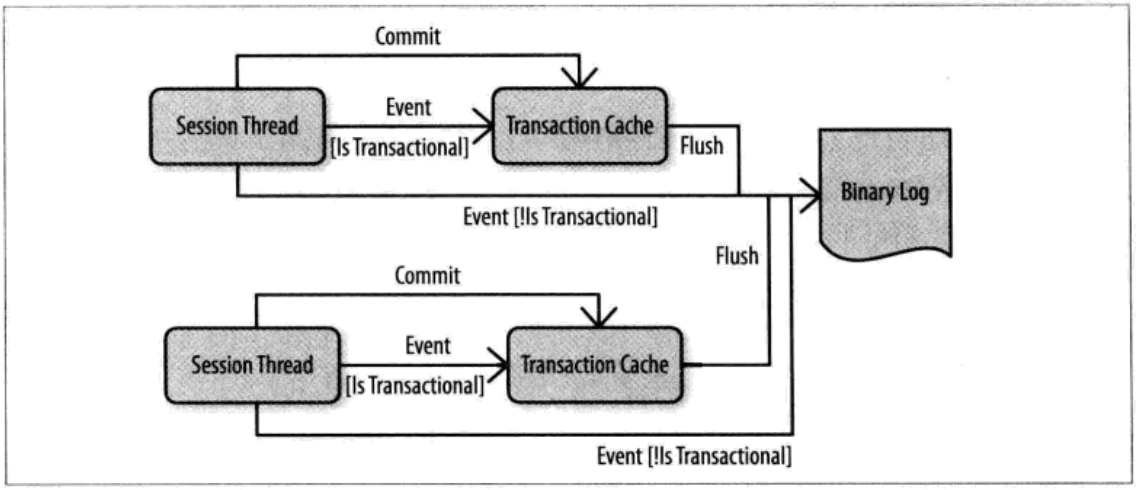


图3-4：事务缓存和二进制日志的线程

需要格外注意包含非事务性改变的语句。回顾一下我们之前的讨论，非事务性语句不会导致当前事务的结束，因此执行非事务性语句引起的变化必须记录在某处，而不必关闭当前活动的事务。当语句同时影响事务和非事务的表时，情况会更复杂。这些语句被看做是事务性的，但是包括了一些不属于该事务的一部分改变。

基于语句的复制不能在所有情况下都正确地处理这些，因此需要采取一个更有效的方法。我们将介绍在服务器上采取的措施，继这个问题之后，你需要知道如何避免遗留的复制问题。

如何记录非事务性的语句

当没有事务是活动的时，非事务性语句会直接被写入二进制日志而不需要在二进制日志结束前“transit”到事务缓存中，然而，如果一个事务是活动的，那么处理语句的规则如下：

1. 如果语句被标记成事务的，它将被写入事务缓存。
2. 如果语句没有被标记成事务性的，而且事务缓存中没有该语句，该语句将被直接写入二进制日志。

78

3. 如果该语句没有被标记成事务性的，但是事务缓存中有语句，则该语句被写入事务缓存。

第 3 条规则可能看起来很奇怪，但如果你看了例 3-14，就会理解其原因了。回到我们的 *employee* 和 *log* 表，考虑例 3-14 中的语句，对事务表的修改在对非事务表的修改之前。

例3-14：非事务性语句的事务

```
1   START TRANSACTION;
2   SET @pass = PASSWORD('xyzzy');
```

```
3   INSERT INTO employee(name,email,password)
    VALUES ('mats','mats@example.com', @pass);
4   INSERT INTO log(email, message)
    VALUES ('root@example.com', 'This employee was bad');
5   COMMIT;
```

根据规则 3，第 4 行的语句将会写入事务缓存，即使这个表是非事务的。如果这个语句直接被写入二进制日志，它将在第 3 行语句前结束，因为直到第 5 行提交成功后，第 3 行语句才会在二进制日志中结束。总之，Slave 的日志最终会在第 3 行实际对 employee 产生改变之前结束，且日志中包含 DBA 在第 4 行添加的注释，这显然与 Master 不一致。规则 3 避免了这样的情况。图 3-5 的左边展示了不应用规则 3 而产生的不良影响，而右边则显示了应用规则 3 而产生的实际情况。

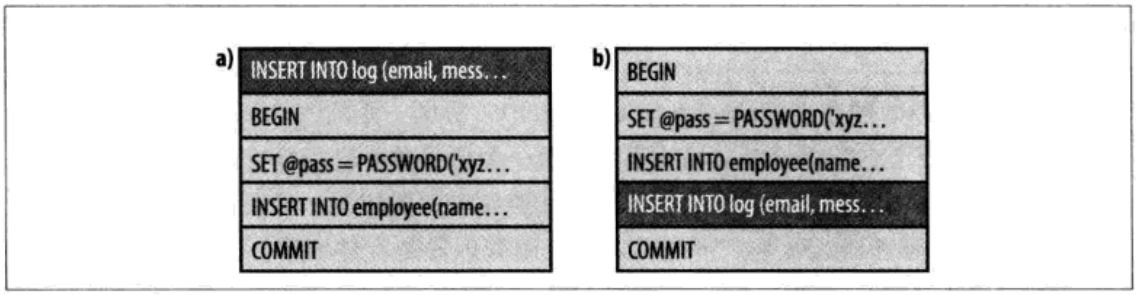


图3-5：可选择的二进制日志随规则3而定

规则 3 涉及权衡。因为当执行事务的时候非事务性语句被缓存，这存在风险：两个事务将以与写入二进制日志不同的顺序更新 Master 上的非事务性表。

当事务中的第一个事务性的语句和第二个非事务性的语句之间有依赖关系时，这个情况将发生，但是这通常并不会被服务器处理，因为它需要完全解析每个语句，包括所有触发器引用的代码，以及执行相关性分析。即使技术上可行，也会增加对一个活动事务中所有语句的额外处理，而且因此将影响性能乃至更多。由于这个问题几乎总是可以通过正确设计事务和确保在事务中没有此类依赖来避免，因此这个开销未增加到 MySQL。

79

如何避免非事务性语句的复制问题

避免在前面章节中讨论的依赖的策略是：确保影响非事务性表的语句在事务中首先被写入。在这个情况下，这个语句将直接被写入二进制日志，因为事务缓存是空的（参考前面章节的规则 2）。该语句是已知没有依赖关系的。

如果在事务的后来，需要从这些语句中得到任何值，可以分配它们到临时表或变量。此后，事物的实际内容将可以被执行，并引用临时表或变量。

使用XA进行分布式事务处理

MySQL 5.0 版本让你可以通过 X/Open Distributed Transaction Processing 模式 XA，协调包括不同资源的事务。尽管目前没有被广泛使用，但 XA 提供了一个诱人的机会来协调事务的各种资源。

在版本 5.0 中，服务器内部使用 XA 来协调二进制日志和存储引擎。

一系列的命令让客户端也可以利用 XA 同步的优势。XA 使不同用户输入的不同语句被当做单个事务来处理。另一方面，它占用了一些开销，因此一些管理员将在全局内关闭它。

虽然 XA 协议的操作指南超出了本书的讨论范围，但在描述它是如何影响二进制日志的之前，我们将在此简单地介绍 XA。

XA 包含一个事务管理器（*transaction manager*），协调一系列的资源管理器（*resource manager*），以便于它们将一个全局的事务当做一个原子单元进行提交。每个事务都被分配一个唯一的 XID，这个 XID 被事务管理器和资源管理器使用。当在 MySQL 服务器内部使用时，事务管理器通常是二进制日志而资源管理器是存储引擎。提交一个 XA 事务的过程如图 3-6 所示，它由两个阶段组成。

80 在第一阶段，每个存储引擎被要求为提交做准备。在准备时，存储引擎将它需要正确提交的一切信息写入到安全的存储器，然后返回一个 OK 消息。如果所有存储引擎的回答都是否定的，则意味着它不能提交这个事务，提交被终止，而且所有的引擎都被通知回滚该事务。

81 在所有的存储引擎都报告它们已经没有问题且准备就绪后，在第 2 阶段开始前，事务缓存被写入二进制日志。普通事务以带有 `COMMIT` 的普通查询事件结束，与此相反，XA 事务则以一个包含 XID 的 `Xid` 事件结束。

在第 2 阶段，在阶段 1 中准备好的所有存储引擎都被要求提交事务。当提交时，每个存储引擎都将报告它已经在稳定的存储器中提交了事务。理解提交不能失败是非常重要的：一旦阶段 1 已经通过，存储引擎就要保证事务能被提交，而且从而不能在阶段 2 报告失败。当然一个硬件错误可以导致系统崩溃，但是由于存储引擎已经在持久的存储器中存储了这些信息，在服务器重启后，它们将能够正确恢复。重启过程将在后面的段落“二进制日志和系统崩溃安全”中讨论。

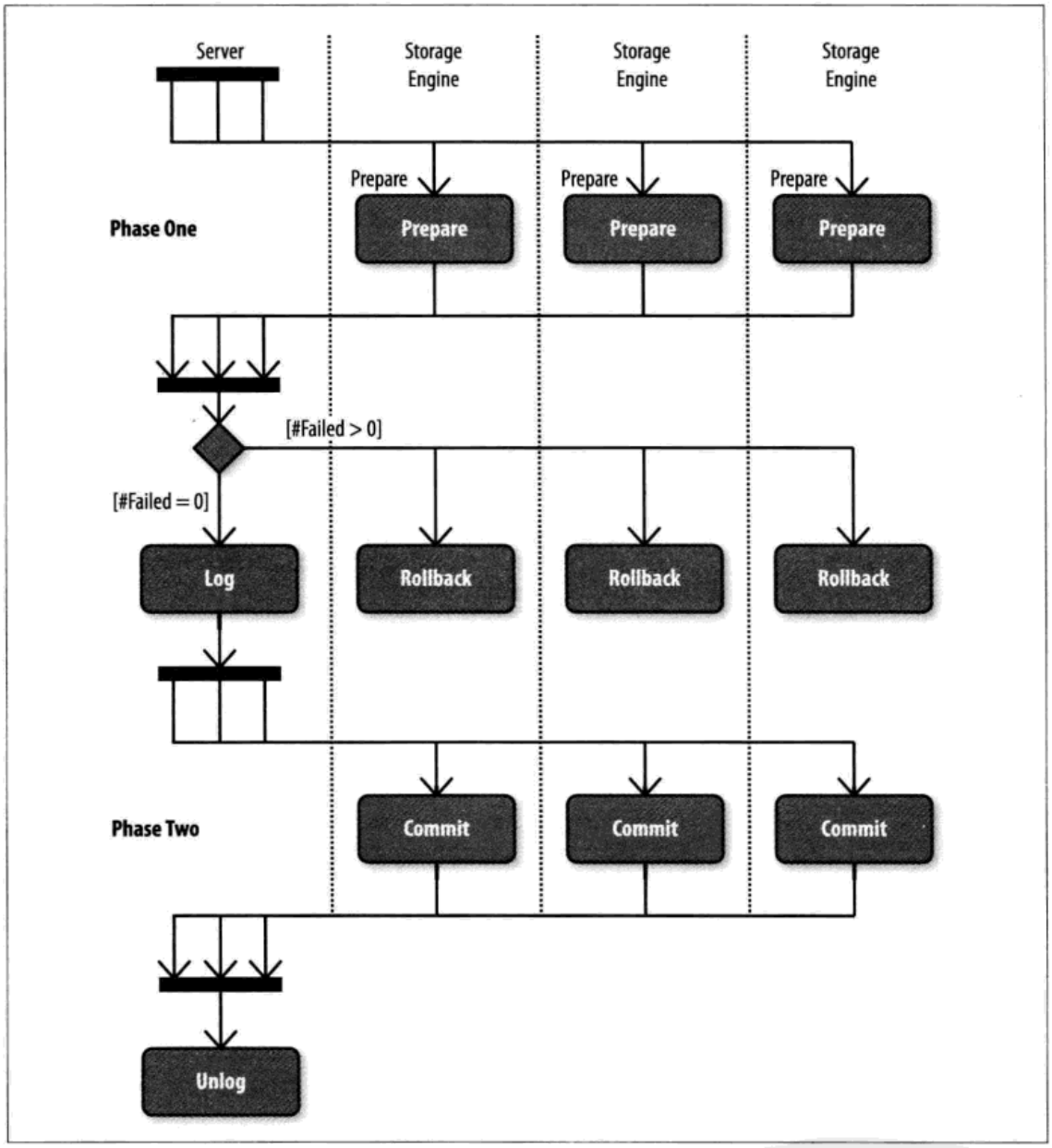


图3-6：使用XA进行分布式事务提交

在阶段 2 之后，事务管理器被赋予一个放弃任何共享资源的机会。二进制日志不需要做任何清理动作，因此在这一步它不需要做任何关于 XA 的特殊处理。

在事件中当执行 XA 事务时系统崩溃发生了，当服务器重启后，图 3-7 中的恢复过程将发生。在启动时，服务器将打开最后的二进制日志并检查 `Format description` 事件。如果前面描述的 `binlog-in-use` 标志被设置，那么它表示服务器已经崩溃而且需要执行 XA 恢复。

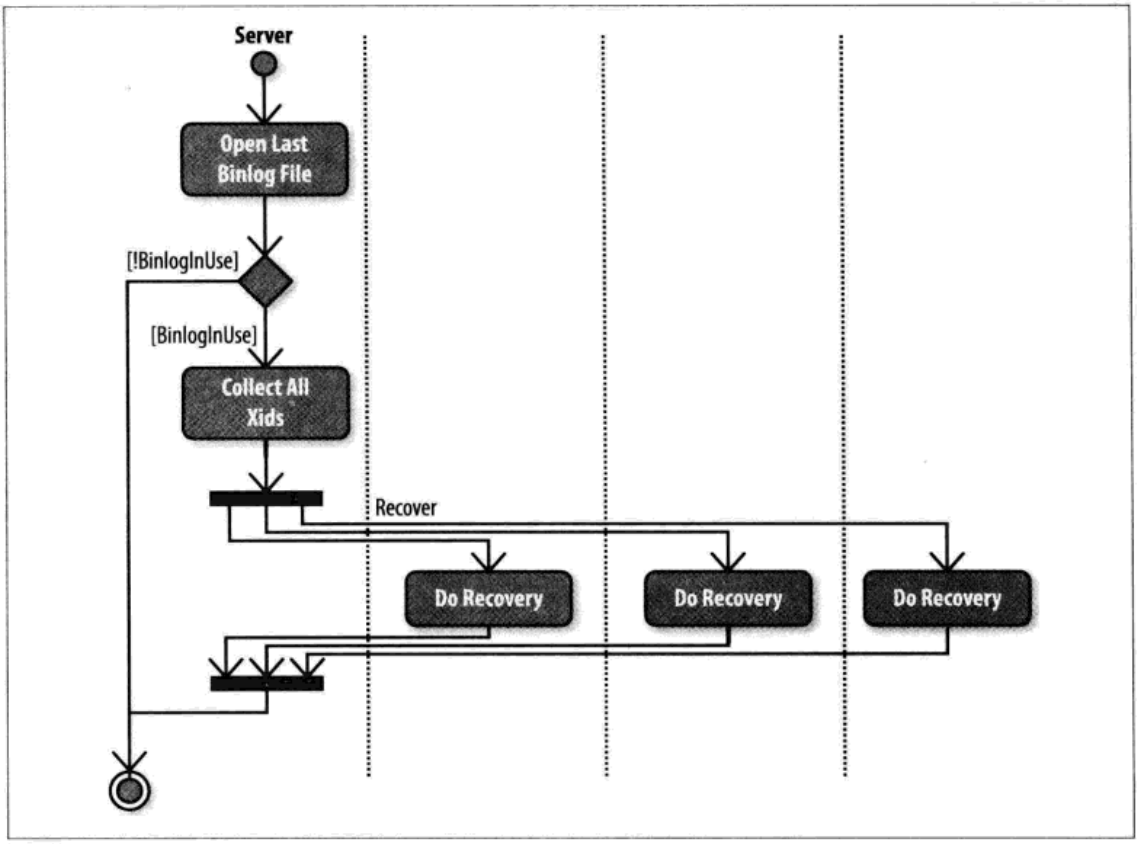


图3-7：XA恢复的步骤

服务器通过走查活动的二进制日志来启动，而且通过读取 `Xid` 事件从二进制日志中查找所有事务的 XID。每个存储引擎装载到服务器后将接着被要求提交列表中的事务。对于列表中的每个 XID，存储引擎将确定哪个 XID 对应的事务是已经准备好但没有提交的，如果是这样就提交这个事务。如果存储引擎已经准备的事务的 XID 不在此列表中，那么这个 XID 显然在服务器崩溃前没有被写入二进制日志，因此这个事务必须被回滚。

二进制日志管理

迄今为止提到的事件代表的都是 Master 上的数据的真实改动，从这个意义上说，它们都是信息的载体。然而，其他事件虽能影响复制，但是并不代表 Master 上数据的改动。例如，如果服务器是停止的，但因在服务器停止时数据文件可以发生改变，它可以潜在地影响复制。一个典型的例子是恢复一个备份，除此之外或是熟练地操作数据文件。这样的改变不会被复制因为服务器没有运行。

82 为其他目的也需要事件。因为二进制日志由多个文件组成，有必要在合适的位置分组，从而组成 binlog 文件序列。为安全地处理，特殊的事件被添加进日志中。

二进制日志和系统崩溃安全

正如你所看到的那样，二进制日志的改变并不相当于基于一对一 Master 数据库的改变。保持数据库和二进制日志相互一致对于防备系统崩溃是非常重要的。换句话说，如果没有写入二进制日志，那么就应该没有更改被提交到存储引擎，反之亦然。

非事务性引擎马上引入了问题。例如，没有必要保证二进制日志和 MyISAM 表之间的一致性，因为 MyISAM 是非事务性的，且该存储引擎将在一切试图记录这语句之前，完成所有更改操作。

但是，对于事务性存储引擎，MySQL 有措施以确保不会因系统崩溃而导致二进制日志失去过多的信息。

正如我们在前面所描述的“记录语句”，事件在释放表上的锁之前、但是在所有的改变已经传送给存储引擎之后被写入二进制日志。因此如果系统崩溃发生在存储引擎释放锁之前，服务器必须确保一切记录到二进制日志的更改在允许语句（或事务）提交前都实际存在磁盘上的表中。

由于磁盘访问比起内存访问花费要高得多，操作系统被设计成这样：缓存部分文件到主内存中的专门部分——当另一个页面必须从磁盘加载而缓存已经满的时候，此时磁盘就变得很有必要了，但它也可以被应用程序要求显式调用：写一个文件的页面到磁盘中。

回顾前面讨论的 XA，当第一个阶段已经完成后，所有的数据被写入持久化的存储器（也就是说，到磁盘），使得协议能够正确处理系统崩溃。这意味着每次当事务被提交的，页面缓存必须被写入磁盘。这将非常昂贵，取决于应用的不同而不是总是必须要这样做。为控制数据多久写入磁盘一次，你可以设置 `sync-binlog` 选项。这个选项用一个整数来指定多久将二进制日志写入磁盘一次。例如，如果这个选项设置为 5，每隔 5 个语句或事务的提交后，二进制日志将写入磁盘。默认值是 0，这意味着二进制日志没显式地被服务器写入磁盘，而是由操作系统决定的。

对于支持 XA 的存储引擎，例如 InnoDB，设置 `sync-binlog` 选项为 1，这意味着对于普通的系统崩溃，你不会丢失任何事务。对于不支持 XA 的引擎，你可能会至少丢失一个事务。

然而，如果每个组都被写入磁盘，这意味着性能变糟，通常会变很糟。众所周知，磁盘访问速度慢，缓存由于不需要总是写数据到磁盘而正好适用于提高性能。如果你愿意冒险失去几个事务或语句，或者因为你可以通过手动恢复来处理这个问题或对于应用程序而言这几个事务或语句不重要，你可以将 `sync-binlog` 的值设置得高一点，或使用默认值。

Binlog文件轮换（Rotation）

MySQL 启动一个新文件来定期保存二进制日志事件。出于实践和管理的原因，始终写到单个文件是不可行的——操作系统对文件大小有限制。正如前面所提到的，服务器当前正在写入的文件是 *active* binlog 文件。

84 切换到新文件时，根据上下文不同，有时叫二进制日志轮换，有时叫 binlog 文件轮换。

有四个主要活动导致轮换。

- 服务器停止
 每次服务器启动，都会开始一个新的二进制日志。不久我们将讨论为什么。
- binlog 文件达到最大尺寸
 如果 binlog 文件增长到很大，它将自动轮换。你可以通过 `binlog-cache-size` 服务器变量来控制 binlog 文件的尺寸。
- Binlog 被显式刷新
 `FLUSH LOGS` 命令将所有日志写入磁盘，然后创建一个新文件用以继续写二进制日志。当为 PITR 管理恢复镜像时，这是很有用的。从活动的 binlog 文件读取可能有预料之外的结果，因此建议在试图使用 binlog 进行恢复前，强制进行显式刷新。
- 一个事故发生在服务器上
 除了全部停止，服务器还可能遇到其他事故而导致二进制日志被轮换。这些事故有时候要求管理员进行特殊的手工干预，因为它们可能在复制流中遗留一个“缺口”。如果遇到事故后，服务器在一个新的 binlog 文件上启动，那么 DBA 就容易处理这个事故。
 每个 Binlog 文件的第一个事件是格式描述（`Format description`）事件，它描述了将文件的内容和状态信息一起写入文件的服务器。
 这里有三个特别有趣的项目。
- `binlog-in-use` *flag*
 因为系统崩溃可能发生在服务器正在写 binlog 文件的时候，关键是当文件已经被正确地关闭时有个标示。否则，DBA 将在 Master 或 Slave 上重放损坏的文件，然后会导致更多的问题。为保证文件的完整性，在最后事件（轮换）被写入文件后，当文件被创建和清除时 `binlog-in-use` 标志会被设置。因此，任何程序都可以看到 binlog 文件是否已经被正确关闭。
- Binlog 文件格式版本
 在 MySQL 的发展历程中，binlog 文件的格式已经改变了多次，而且它一定会再次发生变化。当巨大的更改（对通用头的显著变化）致使服务器之前的版本不能读取新文件时，开发者递增格式的版本号（从 MySQL 版本 5.0 开始，当前格式是版本 4）。

binlog 文件格式的版本字段列出了它的版本号，如果不同的服务器不能处理那个版本的文件，它只是拒绝读取该文件。

- 服务器版本

 用一个字符串表明写该文件的服务器的版本。用于运行本章中的例子的服务器版本是“5.1.37-1ubuntu5-log”，而另一个版本“5.1.40-debug-log”是用来做测试的。如 85
 你所见，这个字符串确保包含 MySQL 服务器的版本，但是它同样包含关于特殊构建（build）的额外信息。在一些情况下，该信息可以帮助开发者找出并解决在不同版本的服务器间进行复制时产生的微小错误。甚至在出现系统崩溃前，为了安全地轮换二进制日志，服务器使用 write-ahead（预写）策略，并在临时文件中记录其意图，该临时文件被称为 *purge index file*（清除索引文件，之所以选择这个名字，是因为正如你将看到的那样，当清除 binlog 文件时，会使用该文件）。它的名字是基于索引文件的，例如，如果索引文件的名称是 *master-bin.index*，清除索引文件（*purge index file*）的名字就叫 *master-bin.~rec~*。在创建新的 binlog 文件之后，再更新指向它的索引文件，服务器再将 *purge index file* 删除。

 在系统崩溃的事件中，如果一个 purge index file 出现在服务器上，当服务器启动后，它可以比较 purge index file 文件和索引文件，看出什么是实际已经完成的而什么是打算做的。



在 5.1.43 之前的 MySQL 版本中，rotation（轮换）或 binlog 文件的清除会遗留一些孤立的文件，也就是说，这些文件可能存在于文件系统中而没有在索引文件中被提及。因此，旧文件没有被正确地清除，在各处留下了它们而且需要手工把这些文件从目录中清除掉。

这些残留文件不会导致复制问题，但它们很可厌。本节展示的程序确保在系统崩溃事件后没有文件被孤立。

事故（incident）

术语“事故”是指在服务器上没有产生数据改变但是必须写到 binlog 的事件，因为它们潜在地影响了复制。大多数事件不需要 DBA 的特殊介入，例如，服务器可以停止和重启而不需要改变数据库文件，但是会不可避免地有一些事件被特殊的操作调用。

当前，你可能在二进制日志中发现如下两种 incident 事务。

- `Stop`

 表示该服务器已通过正常手段停止。如果服务器系统崩溃，即使当服务器再次启动起来，Stop 事件也不会被写入。该事件被写入旧的 binlog 文件（重启服务器会轮换（轮换）到新的文件）而且只包含一个通用头，在事件中没有提供其他信息。

86

当二进制日志在 Slave 上重放时，它将忽略一切 Stop 事件。通常，实际上不需要特别去关注服务器的停止，复制可以照常进行。如果服务器停止时，服务器切换到一个新版本，这将被标志在下一个 binlog 文件中，而服务器如果不能处理新版本的 binlog 文件格式，则将停止读取 binlog 文件。从这个意义上说，Stop 事件在复制流中并不代表是“缺口”。然而，该事件值得记录，因为在重启复制前有人可能会手动地恢复一个备份或者对文件做其他改动，且 DBA 重放该文件将发现这个事件，以在适当的时候启动或停止重放。

- Incident

 版本 5.1 中引入的事件类型作为通用的事故（incident）事件。相比 Stop 事件，该事件包含了一个标识符来指定发生了哪种类型的事故。它用来表示服务器被强制执行那些几乎可以确认从二进制日志中丢失的改变操作。数据库被重新装载或如果一个非事务性事件非常大而不能被 binlog 文件容纳，版本 5.1 中的 Incident 事件会被写入。当 MySQL 集群中的一个节点必须重新装载数据库且因此可能不同步时，MySQL 集群会生成该事件。

 当 binlog 日志在 Slave 上重放，如果遇到 Incident 事件，它会因错误而停止。在 MySQL 集群重载事件的案例中，它表明需要重新同步集群并从 binlog 文件中寻找丢失的事件。

清除binlog文件

随着时间的推移，服务器会积聚 binlog 文件，除非旧的文件从文件系统中清除。服务器可以自动从文件系统中清除旧的二进制日志，或者你可以明确告诉服务器清理这些文件。

要使服务器自动清理旧的 binlog 文件，需设置 expire-logs-days 选项。这个选项也可用来作为一个服务器变量，达到你想保留 binlog 文件的天数。请记住，它就像所有的服务器变量，这个设置不保留在服务器重启之间。因此，如果希望在重启后仍保持自动清除，必须添加该设置到服务器的 *my.cnf* 文件。

使用 PURGE BINARY LOGS 命令手工清除 binlog 文件。该命令有如下两种格式。

- PURGE BINARY LOGS BEFORE datetime

 这个命令格式将清除在给定时间之前的所有文件。如果 *datetime* 在一个日志文件的中间（通常是这样），那么 *datetime* 所在的那个文件之前的所有文件将被清除。
- PURGE BINARY LOGS TO '*filename*'

 这个命令格式将清除在给定文件之前的所有文件。换句话说，SHOW MASTER LOGS 命令输出的所有文件中，在 *filename* 之前的文件都将被删除，使得 *filename* 成为第一个 binlog 文件。

当服务器重启或当二进制日志轮换完成后，binlog 文件将被清除。如果服务器发现需
要清除的文件，无论因为文件是比 expirelogs-days 老或是因为执行了 PURGE BINARY LOGS 命令，当服务器已决定准备清除 purge index file（例如，*master-bin.~rec~*）时，它都将以写文件开始。在此之后，文件将从文件系统中被清除，而且最后清除索引文件（purge index file）也被删除。

在崩溃事件中，服务器可以通过比较清除索引文件（purge index file）和索引文件的内容来继续删除文件，并清除因为系统崩溃而没有被删除的文件，正如你前面所看到的，清除索引文件也会在轮换时使用，因此如果系统崩溃发生在索引文件被正确更新之前，新的 binlog 文件将被删除，然后当再次轮换时被重建。

mysqlbinlog实用工具

对管理员更有用的一个工具是客户端程序 mysqlbinlog。这是一个小程序，它可以审查 binlog 文件及中继日志文件的内容（我们将在第 6 章讨论中继日志文件）。除了在本地读取 binlog 文件，mysqlbinlog 同样可以从其他服务器上远程读取 binlog 文件。

除在调查复制问题时外，它还是一个非常有用的工具，你也可以用它执行 PITR，就像第 2 章中演示的那样。



mysqlbinlog 通常将二进制日志的内容以可执行的格式被发送到运行中的服务器上。当基于语句的复制被使用时，执行语句被当做 SQL 语句发出。对于第 6 章将要介绍的基于行的复制，mysqlbinlog 生成一些处理基于行的复制所必需的额外的数据。本章主要讨论基于语句的复制，因此我们将使用这个命令带一些选项来阻止需要处理基于行的复制的输出。

本章会介绍 mysqlbinlog 的一些选项，如果需要了解完整的列表，请参考在线的 MySQL 参考手册（*http://dev.mysql.com/doc/refman/5.1/en/mysqlbinlog.html*）。

基本用法

下面让我们从一个简单的例子开始。我们创建了一个 binlog 文件，然后用 mysqlbinlog 查看它。我们将启动一个客户端连接到 Master，然后执行下面的命令来看看它们在二进制日志中是如何结束的。

```
mysqld1> RESET MASTER;
Query OK, 0 rows affected (0.01 sec)

mysqld1> CREATE TABLE employee (
      ->    id INT AUTO_INCREMENT,
```

```
      ->    name CHAR(64) NOT NULL,
      ->    email CHAR(64),
      ->    password CHAR(64),
      ->    PRIMARY KEY (id)
      -> );
Query OK, 0 rows affected (0.00 sec)

mysqld1> SET @password = PASSWORD('xyzzy');
Query OK, 0 rows affected (0.00 sec)

mysqld1> INSERT INTO employee(name,email,password)
      ->   VALUES ('mats','mats@example.com',@password);
Query OK, 1 row affected (0.01 sec)

mysqld1> SHOW BINARY LOGS;
+--------------------+-----------+
| Log_name           | File_size |
+--------------------+-----------+
| mysqld1-bin.000038 |       670 |
+--------------------+-----------+
1 row in set (0.00 sec)
```

现在让我们使用 `mysqlbinlog` 来装载这个 binlog 文件 *master-bin.000038* 的内容，这是所有的命令结束的地方。为适应页面显示，例 3-15 所示的输出做了少许编辑。

例3-15：从执行mysqlbinlog得到的输出

```
$ sudo mysqlbinlog         \
>    --short-form          \
>    --force-if-open       \
>    --base64-output=never \
>    /var/lib/mysql1/mysqld1-bin.000038
1  /*!40019 SET @@session.max_insert_delayed_threads=0*/;
2  /*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
3  DELIMITER /*!*/;
4  ROLLBACK/*!*/;
5  use test/*!*/;
6  SET TIMESTAMP=1264227693/*!*/;
7  SET @@session.pseudo_thread_id=999999999/*!*/;
8  SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=1,
       @@session.unique_checks=1, @@session.autocommit=1/*!*/;
9  SET @@session.sql_mode=0/*!*/;
10 SET @@session.auto_increment_increment=1,
89       @@session.auto_increment_offset=1/*!*/;
11 /*!\C latin1 *//*!*/;
12 SET @@session.character_set_client=8,@@session.collation_connection=8,
       @@session.collation_server=8/*!*/;
13 SET @@session.lc_time_names=0/*!*/;
14 SET @@session.collation_database=DEFAULT/*!*/;
15 CREATE TABLE employee (
```

```
16    id INT AUTO_INCREMENT,
17    name CHAR(64) NOT NULL,
18    email CHAR(64),
19    password CHAR(64),
20    PRIMARY KEY (id)
21 ) ENGINE=InnoDB
22 /*!*/;
23 SET TIMESTAMP=1264227693/*!*/;
24 BEGIN
25 /*!*/;
26 SET INSERT_ID=1/*!*/;
27 SET @`password`:=_latin1 0x2A31353141463... COLLATE `latin1_swedish_ci`/*!*/;
28 SET TIMESTAMP=1264227693/*!*/;
29 INSERT INTO employee(name,email,password)
30   VALUES ('mats','mats@example.com',@password)
31 /*!*/;
32 COMMIT/*!*/;
33 DELIMITER ;
34 # End of log file
35 ROLLBACK /* added by mysqlbinlog */;
36 /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

为得到该输出，我们使用三个选项：

`--short-form`

通过这个选项，mysqlbinlog 只打印被执行的的 SQL 语句的信息，而忽略关于二进制日志中的事件的注释信息。当 mysqlbinlog 只是用来回放事件到一个服务器时，这个选项是非常有用的。如果你想为查找问题而审查 binlog 文件，就需要这些注释而不能使用该选项。

`--force-if-open`

如果 binlog 没有被正确关闭，无论因为 binlog 文件仍在被写入或因为服务器系统崩溃，mysqlbinlog 都将打印一条警告说这个 binlog 文件没有被正确关闭。这个选项防止打印那条警告。

`--base64-output=never`

这个选项阻止 mysqlbinlog 打印 base64-encoded 事件。如果 mysqlbinlog 必须打印 base64-encoded 事件，它也将打印二进制日志的 Format description 事件以显示其使用的编码。对于基于行的复制，这并不是必需的，因此该选项被用来阻止那个事件。

在例 3-15 中，1 ～ 4 行包含每个输出打印的序言。第 3 行设置了一个不可能在文件其他地方出现的分隔符。这个分隔符也被设计成在处理语言时作为注释出现，而不被当做分隔符设置。

90

第四行的回滚目的是确保输出不会偶然被放入事务中，因为在输出送到客户端之前，一个事务已经在客户端上开始了。

我们可以暂时跳到输出的末尾——第 33 ～ 35 行——来看看与 1 ～ 4 行相对应的部分。它们修复序言中设置的值，然后回滚所有活动的事务。这是有必要的，因为可以防止 binlog 文件在一个事务中间被截断，而且可以阻止这个输出之后的一切 SQL 代码被包含到事务中。

每当数据库更改时，第 5 行的 use 语句都会被打印。即使 binlog 在每个 SQL 语句前指定数据库，mysqlbinlog 也只显示当前数据库的改变。当一个 use 语句出现时，它就是一个新事件的第 1 行。

确保在每个事件的输出中的第 1 行是 SET TIMESTAMP，正如第 6 行和第 23 行所示。这个语句提供自纪元时间开始以秒计算的事件开始执行时的时间戳。

8 ～ 14 行包含常规的设置，不过类似于第 5 行的 use，只有第一个事件和当它们的值被改变时，它们才被打印。

因为第 29 ～ 30 行的 INSERT 语句是使用用户定义变量插入一张有自增字段的表，第 26 行的会话变量 INSERT_ID 和第 27 行的用户定义变量在语句之前被设置。这是二进制日志中 Intvar 和 User_var 事件的结果。

如果你忽略 --short-form 选项，输出中的每个事件都将被加上一些有关生成该行的事件的注释。你可以看到这些注释，在例 3-16 中它们以 hash 标志（#）开始。

```
$ sudo mysqlbinlog         \
>    --force-if-open       \
>    --base64-output=never \
>    /var/lib/mysql1/mysqld1-bin.000038
   .
   .
   .
1  # at 386
2  #100123 7:21:33 server id 1 end_log_pos 414 Intvar
3  SET INSERT_ID=1/*!*/;
4  # at 414
5  #100123 7:21:33 server id 1 end_log_pos 496 User_var
6  SET @`password`:=_latin1 0x2A313531...838 COLLATE `latin1_swedish_ci`/*!*/;
7  # at 496
8  #100123 7:21:33 server id 1 end_log_pos 643
   Query thread_id=6 exec_time=0 error_code=0
9  SET TIMESTAMP=1264227693/*!*/;
10 INSERT INTO employee(name,email,password)
11   VALUES ('mats','mats@example.com',@password)
```

```
12 /*!*/;
13 # at 643
14 #100123 7:21:33 server id 1 end_log_pos 670 Xid = 218
15 COMMIT/*!*/;
16 DELIMITER ;
17 # End of log file
18 ROLLBACK /* added by mysqlbinlog */;
19 /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

注释的第 1 行给出了事件的字节位置，而第 2 行包含了该事件的其他信息。例如 INSERT 语句那一行：

```
# at 496
#100123 7:21:33 server id 1 end_log_pos 643 Query thread_id=6
    exec_time=0 error_code=0
```

注释的各个部分的意义如下：

- `at 496`
 事件开始的字节位置；也就是说，该事件的第 1 个字节。
- `100123 7:21:33`
 该事件的时间戳 `datetime`（日期加上时间）。这是开始执行查询的时间或事件被写入二进制日志的时间。
- `server_id 1`
 产生该事件的服务器的 server ID。这个 server ID 用来设置 `pseudo_thread_id` 会话变量，如果这个事件是线程特定的而且 server ID 与之前打印的 ID 不同，则设置这个变量的行将被打印。
- `end_log_pos 643`
 紧接着该事件之后的事件的字节位置。通过对比该值和事件开始位置的差异，可以得到该事件的长度。
- `Query`
 事件类型，在例 3-16 中可以看到若干不同的事件类型，例如 `User_var`，`Intvar`，和 `Xid`。

在这些字段之后是 event-specific（事件特定的），且因各个事件而不同。对于查询事件，我们可以看到两个额外的字段：

- `thread_id=6`
 执行该事件的线程 ID。这用来处理 thread-specific（线程特定的）查询，例如访问临时表的查询。
- `exec_time=0`
 以秒为单位的查询执行时间。

92 例 3-15 和例 3-16 转储单个文件的输出，但是 mysqlbinlog 同样也能接受多个文件。如果给定多个 binlog 文件，它们将被按顺序处理。



文件将被按照你要求的顺序打印，而且对以下事件不做任何检查：结束每个文件的轮换事件是否指向序列上的下一个文件。确保这些 binlog 文件是否是真正的二进制日志的一部分是用户的责任。

多亏 binlog 文件的命名方式（例如通过使用 * 作为文件名匹配的通配符），发送多个文件到 mysqlbinlog 通常不是问题。当用来做为文件的扩展名的 binlog 文件计数器从 999999 数到 1000000 时，让我们看看会发生什么：

```
$ ls mysqld1-bin.[0-9]*
mysqld1-bin.000007 mysqld1-bin.000011 mysqld1-bin.000039
mysqld1-bin.000008 mysqld1-bin.000035 mysqld1-bin.1000000
mysqld1-bin.000009 mysqld1-bin.000037 mysqld1-bin.999998
mysqld1-bin.000010 mysqld1-bin.000038 mysqld1-bin.999999
```

正如你可以看到的，最后一个创建的文件列在二进制日志中的次序却在前的两个文件之前。因此在你使用通配符前值得检查一下文件名。

由于 binlog 文件通常都很大，你可能并不希望把 binlog 的整个内容打印出来浏览它们。而有少数选项可以用来限制输出，所以，只有一定范围内的事件被打印。

- `start-position=`*`bytepos`*
 转储的第一个事件的字节位置。请注意，如果几个 binlog 文件是提供给 mysqlbinlog 的，这个位置将被解释为在序列中的第一个文件的位置。
 如果一个事件不在给定的位置开始，mysqlbinlog 仍然会试图解释在那个位置开始的字节作为一个事件，这通常会导致垃圾输出。
- `stop-position=`*`bytepos`*
 最后打印的事件的字节位置。如果没有事件在那个位置结束，最后打印的事件将是位置在 *bytepos* 位置之前的事件。如果给定了多个 binlog 文件，该位置将是序列中最后一个文件的位置。
- `start-datetime=`*`datetime`*
 只打印那些有时间戳或 *datetime* 后的事件。当给定多个文件时，它将正常工作，如果一个文件的所有事件都在 *datetime* 之前，所有的事件都会被略过，而没有检查事件是否按照它们的时间戳顺序打印。

93 - `stop-datetime=`*`datetime`*
 只打印那些有时间戳或 *datetime* 前的事件。这是一个专用的范围，意味着如果一个事件被标记为 2010-01-24 07:58:32 且已经给定了确切的 *datetime*，该事件不会被打印。

请注意，由于事件的时间戳使用语句的开始时间，而二进制日志中的事件是根据提交时间来排序的，那就有可能存在一个事件的时间戳在排在其前面的事件的时间戳之前。由于 mysqlbinlog 可以在时间戳在范围以外的第一个事件处停止，因此有可能不显示这个事件，因为它们的时间戳在 *datetime* 之前。

读取远程文件

除了在本地文件系统读取文件，mysqlbinlog 实用工具还可以从远程服务器上读取 binlog 文件。它是通过使用与 Slave 连接 Master 和请求事件同样的机制来完成这个操作的。这在一些情况下是可行的，因为它不需要在机器上的 shell 账户来读取 binlog 文件，只需要服务器上有一个拥有 REPLICATION SLAVE 权限的用户。

为处理远程读取 binlog 文件，包括 - 含有连接到服务器的主机和用户的 read-from-remote-server 选项，以及可选的端口（如果与默认值不同）和密码。

当从远程服务器读取 binlog 文件时，只需要给出 binlog 文件的名字，而不需要给出它的全路径。

因此为从例 3-16 远程读取 Query 事件，该命令将如下所示（服务器提示输入密码，但当你进入的时候不会显示出来）：

```
$ sudo mysqlbinlog
>    --read-from-remote-server
>    --host=master.example.com
>    --base64-output=never
>    --user=repl_user --password
>    --start-position=386 --stop-position=643
>    mysqld1-bin.000038
Enter password:
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 386
#100123 7:21:33 server id 1 end_log_pos 0 Start: binlog v 4,
    server v 5.1.37-1ubuntu5-log created 100123 7:21:33
# at 386
#100123 7:21:33 server id 1 end_log_pos 414 Intvar
SET INSERT_ID=1/*!*/;
# at 414
#100123 7:21:33 server id 1 end_log_pos 496 User_var
SET @`password`:=_latin1 0x2A3135314146364...38 COLLATE `latin1_swedish_
ci`/*!*/;
# at 496
#100123 7:21:33 server id 1 end_log_pos 643 Query thread_id=6
exec_time=0 error_code=0
```

```
use test/*!*/;
SET TIMESTAMP=1264227693/*!*/;
SET @@session.pseudo_thread_id=6/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=1,
    @@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=0/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_
offset=1/*!*/;
/*!\C latin1 *//*!*/;
SET @@session.character_set_client=8, @@session.collation_connection=8,
     @@session.collation_server=8/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
INSERT INTO employee(name,email,password)
  VALUES ('mats','mats@example.com',@password)
/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

解释事件（Interpreting Events）

有时`mysqlbinlog`打印的标准事件并不足以确定一个问题，因此必须深入事件的具体细节，然后审查它的内容。为处理这样的情况，你可以通过`--hexdump`选项来告诉mysqlbinlog去记录事件的详情。

在查看事件的详情前，请先了解二进制日志中数据格式的一些常规规则：

- 整数数据

 二进制日志的整数字段以Little-Endian顺序打印，因此你必须向后读取整数字段。这意味着，例如，32位的block 03 01 00 00代表16进制数103。
- 字符串数据

 字符串数据通常以长度数据和零值的终端字符串（null-terminated）一起被存储。有时，长度数据只出现在字符串之前而有些时候它被存储在post header里。

本章将涵盖大多数常见的事件，但有关的所有事件的格式细节已经超出了本书的讨论范围。请参看MySQL内部指南（MySQL Internals guide）（*http://forge.mysql.com/wiki/mySQL_Internals_Binaly_Log*），以了解所有可用的事件的完整列表和它们的字段。

所有事件中最常见的就是Query事件，因此我们首先将精力集中在它上面。例3-17展示了这个事件的输出。

例3-17：当使用--hexdump选项时的输出

```
$ sudo mysqlbinlog          \
>    --force-if-open        \
>    --hexdump              \
>    --base64-output=never \
>    /var/lib/mysql1/mysqld1-bin.000038
   .
   .
   .
1  # at 496
2  #100123  7:21:33 server id 1  end_log_pos 643
3  # Position  Timestamp   Type   Master ID       Size      Master Pos    Flags
4  #      1f0 6d 95 5a 4b   02   01 00 00 00  93 00 00 00   83 02 00 00   10 00
5  #      203 06 00 00 00 00 00 00 04 00 00 1a 00 00 00 40 |................|
6  #      213 00 00 01 00 00 00 00 00 00 00 00 06 03 73 74 64 |..............std|
7  #      223 04 08 00 08 00 08 00 74 65 73 74 00 49 4e 53 45 |.......test.INSE|
8  #      233 52 54 20 49 4e 54 4f 20 75 73 65 72 28 6e 61 6d |RT.INTO.employee|
9  #      243 65 2c 65 6d 61 69 6c 2c 70 61 73 73 77 6f 72 64 |.name.email.pass|
10 #      253 29 0a 20 20 56 41 4c 55 45 53 20 28 27 6d 61 74 |word....VALUES..|
11 #      263 73 27 2c 27 6d 61 74 73 40 65 78 61 6d 70 6c 65 |.mats...mats.exa|
12 #      273 2e 63 6f 6d 27 2c 40 70 61 73 73 77 6f 72 64 29 |mple.com...passw|
13 #      283 6f 72 64 29                                     |ord.|
14 #       Query thread_id=6 exec_time=0 error_code=0
   SET TIMESTAMP=1264227693/*!*/;
   INSERT INTO employee(name,email,password)
     VALUES ('mats','mats@example.com',@password)
```

前两行和第 13 行列出了前面讨论的基本信息的注释。请注意，当你使用 `--hexdump` 选项时，一般信息和事件特定的信息被分成两行，而它们被合并在正常的输出中。

第 3 行和第 4 行列出了通用头：

- `Timestamp`
 事件的时间戳（以整数形式表示），以 little-endian 格式存储。
- `Type`
 单个字节代表事件的类型。MySQL 内部指南中给出了 MySQL 版本 5.1.41 和之后的版本中的事件类型（*http://forge.mysql.com/wiki/MySQL_Internals_Binary_Log*）。
- `Master ID`
 写该事件的服务器的 server ID（以整数形式表示）。对于例 3-17 中的事件，server ID 是 1。
- `Size`
 以字节为单位的事件大小（以整数形式表示）。
- `Master Pos`
 同 `end_log_pos` 类似，也就是说，接着这个事件之后开始的事件。

96 • Flags

该字段用 16 位来保存跟事件相关的常规标志。这个字段大多数情况下是不使用的，但是它存储了 binlog-in-use 标志。正如你在例 3-17 所看到的，binlog-in-use 被设置，意味着二进制日志没有被恰当地关闭（在这个案例中，是因为我们没有在调用 mysqlbinlog 之前刷新日志）。

在通用头之后是提交头（post heade）和事件体。正如已经提到过的，所有事件的完整列表已超出了本书的讨论范围，但是我们将涵盖最重要、最常用的事件：Query 和 Format_description 日志事件。

查询事件的提交头和体

查询事件是目前由服务器发出的最常用也最复杂的事件。一部分原因是它必须承载已被执行的语句内容的大量信息。正如已经表明的，已经使用特定事件覆盖了整数变量、用户变量和随机种子，但是仍有必要提供作为这个事件的一部分的其他信息。

Query 事件的 post header 由 5 个字段组成。回想一下，这些字段都是固定大小的，而 post header 的长度是在 binlog 文件的 Format deseription 事件中给出的，这意味着如果有需要，以后的 MySQL 版本可能会增加额外的字段。

- *Thread ID*

 一个四字节无符号整数表示执行该语句的 thread ID。尽管 thread ID 并不是正确执行语句所必需的，但是它始终都被写入事件。
- *Execution time*

 从开始执行查询到它被写入二进制日志所花费的秒数，以一个四字节的无符号整数表示。
- *Database name length*

 数据库名的长度，存储为一个无符号单字节整数。数据库名被存储在事件体中，但是长度在这里给出。
- *Error code*

 从语句的执行结果得到的错误代码，存储为两个字节的无符号整数。此字段被包含在内是因为在某些情况下，即使它们失败了，语句也必须记录到二进制日志中。
- *Status variables length*

 事件体中存储状态变量的块长度，存储为两个字节的无符号整数。这个状态块有时和 Query 事件一起用来存储各种状态变量，例如 SQL_MODE。

97 事件体包括以下字段，它们都是可变长度的。

- *Status variables*

 状态变量序列。每个状态变量都以单个整数接着状态变量的数值来表示。每个状态变量值的解释和长度都取决于其关注的变量是哪个：状态变量不总是存在的，它们只在必要时添加。一些状态变量的例子如下：

 - `Q_SQL_MODE_CODE`

 当执行语句时使用 `SQL_MODE` 的值。

 - `Q_AUTO_INCREMENT`

 这个状态变量包含语句中使用的 `auto_increment_increment` 和 `auto_increment_offset` 的值，假定它们的默认值不为 1。

 - `Q_CHARSET`

 这个状态变量包含在执行该语句时连接和服务器所使用的字符集代码和排序规则。

- *Current database*

 当前数据库的名称，存储为一个空结束符（null-terminated）的字符串。请注意，数据库名称的长度在 post header（提交头）中给定。

- *Statement text*

 被执行的语句。语句的长度可以从通用头和提交头的信息中计算出来。这个语句通常与原始记录的语句相同，但是在某些情况下，语句在它被存储到二进制日志前被重写。例如，正如你在本章前面所看到的，触发器和存储过程与指定的 `DEFINER` 从句一起被存储。

格式描述事件提交头和体

`Format_description` 事件记录了关于 binlog 的文件格式、事件格式和服务器的重要信息。由于它必须在各个版本之间保持健壮（即使 binlog 格式变化，它仍必须是可以解释的），所以对允许哪些改变有些限制。

最重要的限制之一是 `Format_description` 事件和 `Rotate` 事件的通用头都是固定的 19 个字节。这意味着没有必要在通用头中增加新字段来扩展该事件。`Format_description` 事件的提交头和事件体包含如下的字段：

98

- *Binlog file version*

 文件所使用的 binlog 文件格式的版本。对于 MySQL 版本 5.0 及后来的版本，它是 4。

- *Server version string*

 一个 50 字节的字符串存储了服务器版本的信息。通常由三部分组成：版本号，接着是 build 使用的选项信息，例如“5.1.37-1ubuntu5-log”。

- *Creation time*

 服务器自启动后写入的第一个 binlog 文件的创建时间，一个四字节的整数（自从

UNIX 纪元（格林威治时间 1970 年 1 月 1 日 00:00:00）以来的秒数）。对于之后服务器写入的 binlog 文件，这个字段将是 0。

这项设计让一个 Slave 确定该服务器已重新启动，而 Slave 应该重置状态和临时数据。例如，关闭一切活动的事务及删除所有已经创建的临时表。

- *Common header length*

 在 binlog 文件中除了 `Format_description` 和 `Rotate` 事件之外的所有事件的通用头长度。正如前面所描述的，`Format_description` 和 `Rotate` 事件的通用头的长度是固定的 19 个字节。

- *Post-header lengths*

 这是 `Format_description` 日志事件的唯一可变长度的字段。它拥有一个数组，binlog 文件中的每个事件的 post header 的大小作为一个字节的整数被包含在其中。该字段的长度为 255，因此 post header 的最大长度是 254 字节。

二进制日志选项和变量

一系列的选项和字段使你能去配置二进制日志的诸多方面。

一些选项控制 binlog 文件和 index 文件名称这样的属性。大多数选项也可以被按照服务器变量处理。有一些已经在本章的前面提到了，这里将介绍每个选项的更多细节信息：

- `expire-log-days=`*`days`*

 binlog 需要保留的天数。当二进制日志已经轮换或服务器重启时，比指定数值老的文件将从文件系统中清除。

 这个选项默认是 0，意味着 binlog 文件永远不会被删除。

99

- `log-bin[=`*`basename`*`]`

 通过在 *my.cnf* 文件中添加 `log-bin` 选项来开启二进制日志，正如在第 2 章所解释的。除了打开二进制日志，这个选项还给出了 binlog 文件的基本名称，也就是在圆点之前的部分文件名。如果提供了扩展名，当组成 binlog 文件的基本名称时，它将被移除。如果该选项没有指定 *basename*，则基本名称默认为 *host*`-bin`，而 *host* 就是基本名称（也就是说，没有目录和扩展名的文件名）由 pid-file 选项指定的文件名，通常 hostname 由 gethostname(2) 指定。例如，如果 `pid-file` 是 */usr/run/mysql/master.pid*，默认的 binlog 文件名将是 *master-bin.000001*，*master-bin.000002* 等。

 由于 `pid-file` 选项的默认值包含 hostname，强烈建议你给 `log-bin` 选项赋值。否则当 hostname 改变时，binlog 文件将改名（除非 `pid-file` 被给定一个确切值）。

- `log-bin-index[=`*`filename`*`]`

 给出索引文件的名称。如果你想放置索引文件到不同于默认位置的其他位置，它可

能有用。

默认与 log-bin 所用的基本名称相同。例如，如果用来创建 binlog 文件的基本名称是 *master-bin*，索引文件将被命名为 *masterbin.index*。

与 log-bin 选项情况类似，hostname 将被用来组成索引文件名，意味着如果 hostname 改变，复制将中断。出于这个原因，强烈建议你给这个选项赋值。

- log-bin-trust-function-creators

 当创建存储函数时，可以创建特制的函数，允许在 Slave 上进行任意的数据读取和处理。为此，创建存储函数需要 SUPER 权限。但是，由于存储函数在很多情况下是非常有用的，这也许是 DBA 信任一切拥有 CREATE ROUTINE 权限的人不会写恶意存储函数。因此，对于创建存储函数，它可以禁止对 SUPER 权限的需求（但 CREATE ROUTINE 仍然需要）。

- binlog-cache-size=*bytes*

 以字节为单位的事务缓存的 in-memor 部分的大小。事务缓存通过磁盘备份。因此当事务缓存的大小超过该值时，剩余的数据将进入磁盘。

 这有可能造成性能问题，因此如果你使用许多大型的事务，增大该选项的值可以提高性能。

 注意，只分配非常大的缓冲区大小并不是个好主意，因为这意味着服务器的其他部分得到更少的内存，这可能导致性能下降。

- max-binlog-cache-size=*bytes*

 使用这个选项来限制在二进制日志中的每个事务的大小。由于大型事务有可能阻塞二进制日志很长时间，它们将导致其他线程为二进制日志护航而造成重大的性能问题。如果事务的大小超过 *bytes*，该语句将出错而被终止。

- max-binlog-size=*bytes*

 指定每个 binlog 文件的大小。当写入语句或事务将超过这个值时，binlog 文件将被轮换且写入一个新的空的 binlog 文件。

 请注意，如果事务或语句超过了 max-binlog-size，二进制日志将被轮换，但该事务的全部内容将被写入新的文件，超过了指定的最大值。这是因为事务永远不会被分割到不同的 binlog 文件。

- sync-binlog=*period*

 指定多长时间通过 *fdatasync(2)* 写二进制日志到磁盘一次。给定的值是每次真实调用 *fdatasync(2)* 的事务提交数量。例如，如果给定数值为 1，每次事务提交时调用 *fdatasync(2)*，而如果给定值为 10，则每 10 个事务提交将调用一次 *fdatasync(2)*。

 数值为 0 表示将永远不会调用 *fdatasync(2)*，而且服务器相信操作系统会写二进制日志到磁盘，作为正常文件处理的一部分。

- `read-only`
 防止任何客户端进程（除了 Slave 进程和有 SUPER 权限的用户）更新服务器上的任何数据。这对于 Slave 服务器让数据在不被连接到 Slave 的客户端毁坏的情况下处理复制是非常有用的。

小结

显然，二进制日志有很多要说的，包括它的使用、组成和技术。在本章中介绍了这些概念和更多的内容，包括如何控制二进制日志的行为。本章介绍的内容构建了对二进制日志进行更深入了解的基础。本章还介绍了记录数据更改的重要性。

101

Joel 打开一封老板发来的无标题的邮件。“我讨厌人们这样做。”他想。Summerson 先生的邮件就像他的任务分配——闲话少说，进入正题。该邮件写道：“谢谢你为营销人员恢复数据。我将期待明天上午的报告。你可以通过电子邮件发送报告。”

乔尔耸耸肩，开始创建新的电子邮件信息，并认真地写入了有意义的邮件主题。他想想应该包括什么层次的细节，以及是否需要解释他所了解的二进制日志和 `mysqlbinlog` 实用工具。沉思片刻之后，他写下了所能包含的许多细节。“他很可能会让我把它削减成一个项目符号列表。”Joel 想。这似乎是个好主意，于是他写了两句话的总结和一些要点，并将它们移到该邮件的顶部。当他完成以后，把这个邮件发送给了老板。“也许我应该开始在某处保存这些，以防止万一我不得不讲述某事。”他自言自语道。

基于复制的数据库高可用技术

Joel 正在听他的 iPod，一抬头发现老板就站在他的办公桌前。他摘下耳机说："对不起，老板。"

Summerson 先生微笑着说："没关系的，Joel。我需要你帮忙找出一个可以持续监控复制服务器的办法，既能保证数据不会丢失，又能减少服务器停机时间。我们已经听到一些开发人员抱怨当前系统不是很灵活。我和他们一起检查过，技术支持人员告诉我，当服务器出点儿错后，总是要花很长时间才能恢复。我希望你现在首要解决这个问题。"

Joel 点了点头，"当然，我会检查一下负载平衡，并改善复制服务器的恢复能力。"

"太好了！看看解决这个问题我们需要做些什么，给我出份报告。"Joel 目送他的老板离开办公室。"好吧，让我们看看"基于复制的数据库高可用技术"这一章能告诉我们什么。"Joel 一边想，一边打开了他最爱的《高可用 MySQL》这本书。

我们购买可靠的昂贵机器，并确保拥有一个良好的 UPS（不间断电源，Uninterrupted Power Supply）以避免系统断电，在这种情况下，应该可以得到一个高可用的系统，对吧？

然而，高可用性其实并不是那么容易就能实现的。要想搭建一个真正的持续可用的系统，你必须仔细地考虑到任何偶然故障的发生，并确保有冗余故障处理组件。真正的高可用性是指系统即使处在最意想不到的环境中都不会停止运行，这样的高可用性系统是非常难以实现的，并且花费也是很昂贵的。

实现高可用性的规则非常简单，只需要记住三点：

- 冗余

 如果一个组件出现故障，你必须有一个可替代品。该替代品既可以是闲置的，也可以是当前系统部署中的一部分。

104 >
- 应急计划

 如果一个组件出现故障，你必须知道接下来该做什么。这取决于哪个组件出现故障，以及为何出现故障。

- 程序

 如果一个组件出现故障，你必须能够检测出故障原因，然后迅速有效地执行你的计划。

一个系统中即使只有一个组件出现故障，也将会导致整个系统瘫痪——它严重限制了我们实现高可用性的能力。这意味着首要目标之一，就是要找到这些单个的故障点，并确保已经提供了冗余故障处理。

冗余

为了搞清楚哪里需要冗余机制，必须明确系统部署中每一个可能出现的故障点。虽然这听起来容易，但它需要一定的想象力，以确保真正找到它们。交换机、路由器、网卡甚至通信电缆都可能会是故障点。架构当然也很重要，除此以外就是电源和物质设施了。但是，如何保证系统部署的后续服务？假设所有的网络管理都被合并到一个基于 Web 的接口，或者，如果你只有一名工作人员知道如何处理某些类型的故障怎么办？

确定故障点并不意味着你必须完全消除它们。有时，从经济、技术、地理等因素考虑，完全消除也是不可能的，但是知道它们的存在有助于我们实施计划。

有些事情应该考虑到，或者，至少有意识地决定是否该考虑复制组件的成本、不同组件发生故障的概率、替代故障组件的时间和修复组件的风险。如果需要一周的时间才能替换一个组件，而且在这个故障发生时，备用组件正在运行，那么你需要承担备用组件也会丢失的风险，这可能是可接受的，也可能是不可接受的。

一旦确定哪里需要冗余，我们必须从两个基本方案中选择：既可以在每个组件周围都保留备份，一旦原先的组件发生故障，备份都可以立刻派上用场；也可以确保系统有额外能力，以便于一个组件出现故障时，依然可以处理其负载。两个方案并不是只能选择一个：你可以结合两种方案，备份一些组件，并利用系统其他部分的额外能力。

从表面上看，最简单的方法是备份组件，但是备份的耗费是很大的。你必须让闲置的备份组件平时处于随时待命状态，而且需要一直与主要组件一起更新。备份组件的好处是，
105 > 当用备份组件替换故障组件时系统依然保持良好性能，切换到备份组件比重组系统要快很多，如果创建备用容量时遇到问题，你也只能选择重建系统。

创建备用容量可以让你利用整个组件来运行业务，也可以让你处理更高的负载。当一个组件停止运行时，重组系统能保证余下的所有组件可以正常运行。然而，拥有比平时所需更多的容量很重要。

为了了解原因，我们考虑一个简单的例子，有一个处理写任务的 Master，实际上，你应该有两个 Master，因为你需要有冗余；一系列 Slave 连接到 Master，这些 Slave 的唯一目的就是处理读请求。

如果一个 Slave 失败了，系统仍会响应，但系统的容量就变小了。如果你有 10 个 Slave，那么每个 Slave 的运行负载是 50%，其中一个 Slave 的失败会让每个 Slave 的负载增加到 55%，这还在能接受的范围之内。然而，如果每个 Slave 的运行负载是 95%，那么任何一个 Slave 的失败会使剩余的 Slave 的运行负载增加到 105%，这显然是不可能的。在这种情况下，系统的读入能力将会降低，响应时间也会更长。

然而只对一台服务器发生故障做出应对计划还是不够的：你需要考虑到多台服务器同时发生故障的可能性和其应对方法。继续之前的例子，即便每台服务器的运行负载是 80%，这个系统还能应付一台服务器发生故障的情况。然而，两台服务器都发生故障，意味着当前剩余的每个服务器的负载将上升到 100%，这将让你没有任何能力再应对其他任何突发事件了。如果这种几率可以限定为一年发生一次，或许还是可以处理的，但是你必须知道它发生的频率有多高。

表 4-1 显示了在安装 100 台服务器的系统中，1 台、2 台、3 台服务器发生故障的不同概率。正如你所看到的，一台服务器发生故障的概率是 1% 时，三台以上服务器发生故障的概率是 16%。如果你没有为处理这种情况做任何准备，那么一旦该种情况发生，你将措手不及。



随机变量 X 代表发生故障的服务器的数量，通过二项式定理计算得到概率。

$$P(X \geq k) = \binom{n}{k} P^k$$

表4-1：服务器发生故障的概率

单个服务器发生故障的概率	1	2	3
1.00%	100.00%	49.50%	16.17%
0.50%	50.00%	12.38%	2.02%
0.10%	10.00%	0.50%	0.02%

106 为了避免这种情况，你必须时刻监视部署系统以了解系统的负载情况。通过测量计算出系统容量，并利用数学计算查看哪里的响应时间开始受到影响。

计划

只做冗余计划是不够的，当一个组件发生故障时，你需要对接下来该怎么办做出应对策略。在之前的例子中，当一个服务器发生故障时是很容易处理的，因为新的连接将会被重定向到正在工作的服务器，但是要考虑以下几点：

- 对已存在的连接将会发生什么？如果只是终止运行并将错误直接抛给用户，这并不是一个好主意。通常情况下，在用户和数据之间有一个应用层，所以在这种情况下，用户层必须重试与另外一个服务器连接查询。
- 如果 Master 发生故障了怎么办？在先前的例子中，都是假设 Slave 发生故障，但是 Master 也是很有可能出故障的。假设你已经提供了一个额外的 Master 做冗余（我们将在后面的章节中介绍怎么做），你必须提供所有的服务器都会连接到新的 Master 的机制。

本章将会介绍一些可以用来处理 MySQL 服务器发生故障的各种情况的技术和拓扑结构。基本上有三种角色的服务器需要考虑：Master 故障，Slave 故障和 relay 故障。就 Slave 故障而言，只有超出读范围的 Slave 才会发生故障。这些同时扮演 Master 角色的 Slave 都是中继服务器，它们需要特别的照顾。Master 故障是需要最快处理的重要故障，因为系统直到 Master 恢复才能运行。

Slave故障

到目前为止，最容易处理的故障是 Slave 故障。因为这些 Slave 都只是用于读查询，它足以向负载平衡器通知哪台服务器发生故障了，负载平衡器会将新的查询递交到正在工作的服务器。当然必须有足够多的 Slave 去处理系统能力的降低，但除此之外，一个发生故障的 Slave 一般不会影响到复制的拓扑结构，也不需要考虑特别的拓扑结构以保证发生故障的 Slave 易于管理。

当一个 Slave 发生故障时，难免有一些查询已经发送到这个 Slave 并在等待查询结果。一旦这些连接从一个已经失去连接的服务器中返回错误报告，这些查询将会重新递交给正在工作的 Slave。

Master故障

如果 Master 发生故障，它必须能被替换以保证部署系统可以正常运行，而且这种替换也

必须是迅速的。当 Master 发生故障的那一刻，所有的写查询都会被中断，所以第一件要做的事就是得到一个新的可用的 Master，并将所有的客户端连接到这个新的 Master 上。

由于 Master 发生故障，所有的 Slave 都失去与 Master 的连接，这意味着所有的 Slave 都有过时的数据，但是它们仍在运行并回复读查询。

然而，如果有些查询正在监听到达 Slave 的变化，那么这些查询会被阻止。同时这些查询会把这一情况记录到 Slave 的中继日志中，因此最终将由 Slave 执行查询。不需要专门考虑这些查询的行为。

如果查询正在等待 Master 上运行的事件的结果时，Master 发生了故障，这时情况将会变得更加糟糕。在这种情况下，有必要去确认这些事件已经被处理了。而且这也通常意味着查询被报告为失败，因此用户必须重新发出查询请求。

中继服务器故障

处理作为中继服务器的服务器故障的情况比较特殊。如果它们发生了故障，其他的 Slave 必须重新连接到其他中继服务器或 Master 上。由于添加中继服务器就是为了减轻 Master 的负载，有可能出现以下情况：Master 不能处理连接到它的一个中继服务器上的批量 Slave 的负载。

灾难恢复

在高可用性的世界里，“灾难”并不意味着地震或者洪水，它只是表明电脑出了极坏的状况，不仅仅是局部出了问题。

典型的例子是数据中心断电了——不一定是因为整个城市断电了，只是数据中心所在的地方断电了。请参考“mysql.com 断电期”一节，了解一个实际发生过的事件，并看看这时 MySQL 究竟发生了什么状况。

灾难的本质在于，很多故障同时发生，就使得在单个数据中心，备份服务器无法处理冗余。由此可见，有必要确保数据被安全保存在另外一个安全地方，除此之外，很多公司经常采取的措施是，在不同的办公室存放不同的组件，即便是相对较小的公司也会这么做。

程序

在你已经排除了所有的单点故障后，确保系统有足够的冗余，并为每个紧急事件做好了紧急措施，你应该为最后一步做好准备。

除非你能恰当地运用它们，否则你所有的资源和精心准备都是无用的。对一个只有几个服

务器的小型网站，你经常可以没有规划，并手动管理这些服务器，但是随着服务器数量的增加，自动部署将变得很有必要——而且如果你的商业运作成功，服务器的数量也必然快速增加。

108 如果你从一开始就为自动化做准备，那么你会感觉好很多——如果你的公司业务增长，你将会忙于处理其他的事情，可能没有时间去建立必要的自动化支持。

我们已经讨论过一些基本的程序，但是你至少必须考虑以下任务的备用程序：

- 增加新的 *Slave*

 在扩大网站的规模时创建新的 Slave 是运行大规模网站的基础。创建新的 Slave 的方法很多。它们都采取了围绕方法（circle around methods），对现有的服务器（通常是 Slave）进行快照，紧接着在一个新的服务器上恢复快照，然后从正确的位置开始复制。

 当然，对新服务器采取快照的时间，将会影响到你增加新 Slave 的时间。如果备份时间太长，Master 上可能会发生很多新的更改，这意味着新的 Slave 会花很长时间同步。出于这个原因，快照的时间很重要。图 4-1 显示了 Slave 同步时的快照时间。可以看出，当 Slave 停下来去执行快照时，变化将开始累积，这样将导致未解决的更改增加。一旦 Slave 重启，它会开始响应未解决的更改，未解决的更改的数量将减少。

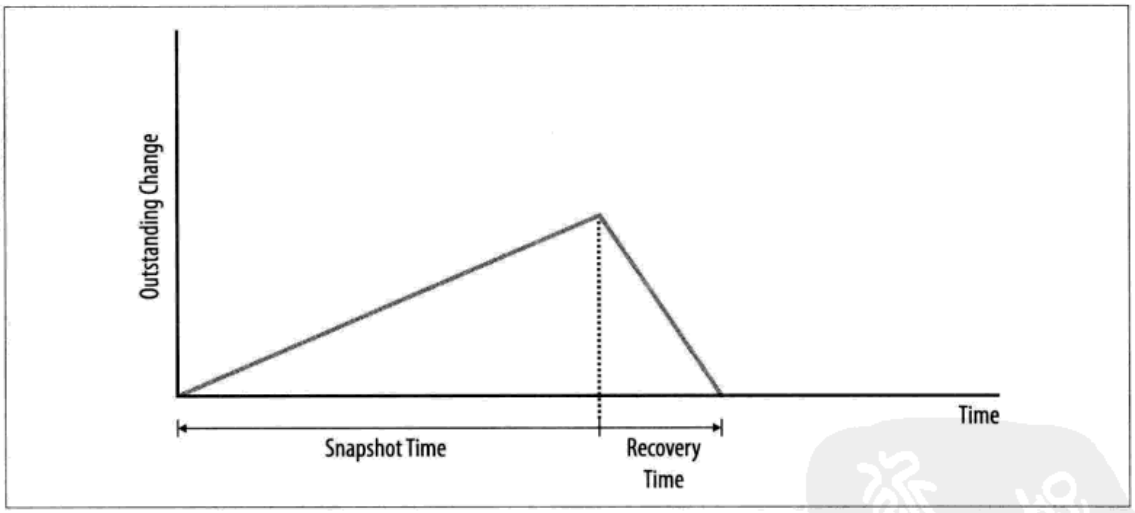


图4-1：获取快照时的显著变化

一些不同的快照方法如下所示：

- 使用 `mysqldump`

 使用 `mysqldump` 是安全的，但是速度很慢。它允许你用与先前不同的存储引擎去存储数据，如果你使用 InnoDB 表，可以得到一致性快照，意味着你不需要脱机进行快照。

- 复制数据库文件

 这个方法相对比较快，但是需要脱机复制文件。
- 使用在线备份方法

 有不同的可用方法，例如 InnoDB Hot Backup。
- 使用 *LVM* 获取快照

 在 Linux 系统中，可以通过逻辑卷管理器（LVM）得到卷快照。它要求你事先准备创建一个特殊的逻辑卷管理器。
- 使用文件系统快照的方法

 例如，Solaris ZFS 支持内置快照。这是创建备份的最快的技术，但是和上面提到的其他技术相似（`mysqldump` 除外）。这种方法意味着，你不能重建一个使用不同引擎的服务器。

 创建新的 Slave 技术已经在第 2 章中提到，不同的备份方法将在第 12 章中介绍。 109
- 从拓扑结构中删除 Slave

 从设置中删除 Slave 只需要通知负载均衡器删除的 Slave 已经不存在。可以在第 5 章找到一个负载平衡器的例子，该例子中包括了添加和删除服务器的方法。
- 切换 Master

 对于日常维护，将连在 Master 上的所有 Slave 都切换到备份 Master 是很正常的，同时要通知负载均衡器：旧 Master 不存在。这个过程可以而且应该无须停机处理，因为它应该不影响正常的操作。

 使用 Slave 提升（本章后面会描述）是解决这个问题的一个办法，但是使用热备份替代可能更容易（也会在本章后面描述）。
- Slave 故障处理

 Slave 会出现故障——这只是频率问题。处理 Slave 故障在任何的部署中都必须当成一个常规事件来看。只需检测到 Slave 不在，然后把它从负载平衡池中移除，这将在第 5 章中进行描述。
- Master 故障处理

 当一个 Master 突然出现故障时，你必须能检测到这个故障，并把所有的 Slave 都连接到一个备用 Master 上，或者临时从子服务中提升一个新的 Master。本章的稍后部分将描述这些技术。
- 更新 Slave 110

 将 Slave 升级到新版本通常应该不会出现问题。但是由于要升级 Slave，需要将它从系统中移除，且需要将它从负载均衡器中移除，并需要通知其他系统这个 Slave 不存在了。
- 升级 Master

 为了升级 Master，经常有必要首先升级所有的 Slave。然而，情况也并非全部如此。

为了升级 Master，当你在执行升级时可以使用备份 Master，或提升其中一个子服务器成为 Master。

mysql.com 停机事件

MySQL 的 IT 团队是由一组多才多艺且非常敬业的队员构成的，他们能够处理各种系统和设备。与我这么多年见过的许多其他 IT 团队不同，这些家伙们都能轻松自在地处理 MySQL 这么多年积累的一系列复杂的计算机排列（从高端的 Windows 机器到非常古老的 SGI Irix 和 HP-UX 机器），不管用什么样的排列把它们都聚集在一起。

对于在许多不同的机器上测试 MySQL 服务器而言，数据中心是非常宝贵的，但是随着 MySQL 服务器的增加，那里开始变得非常拥挤。所以，IT 团队在斯德哥尔摩建立了一个更好、更贵的数据中心。这个搬迁计划在周末进行，但是在实施计划前事情发生了严重的转变。

我经常在家工作，但在这个特殊的日子，我需要去位于乌普萨拉的办公室开一些项目会议。那天早上我发现 mysql.com 的站点挂掉了，但是我依然离开了，只是往好的方面去想。

到达乌普萨拉的办公室，我发现 MySQL IT 团队的部分人员将电源线都串在一起，并将这些电源线连接到数据中心。显然，整栋大厦已经断电了，但是临近的建筑物都还有供电，所以 IT 团队已经串起一条长的电源线来给重要的开发服务器供电。

停电是相当严重的，而且当我赶到时 UPS 也已经耗尽了。向大厦供电的电网已经被破坏，电力公司的工程师也不知道什么时候能解决供电问题，部分 IT 队员已经决定立刻转移网堆栈（web stack），他们基本上是用能找到的车把网络协议栈的机器搬迁到斯德哥尔摩以南约 100 公里的新数据中心。

网堆栈（web stack）在斯德哥尔摩上线后，mysql.com 网站被恢复，但是在电网修好之前，需要做大量的工作去恢复开发服务器，利用一切可利用的电源。这个团队无休无眠地工作了 48 个小时后机器才完全恢复了运行——对于 IT 团队，真是无话可说。

MySQL 重新回到轨道上……

111

热备份

最简单的服务器备份技术就是热备份技术。如图 4-2 所示，该热备份拓扑结构由 Master

和一个称为热备份的专用服务器组成，热备份服务器从 Master 中复制数据。热备份服务器像 Slave 一样连接到 Master 上，它读入并应用所有的更改。

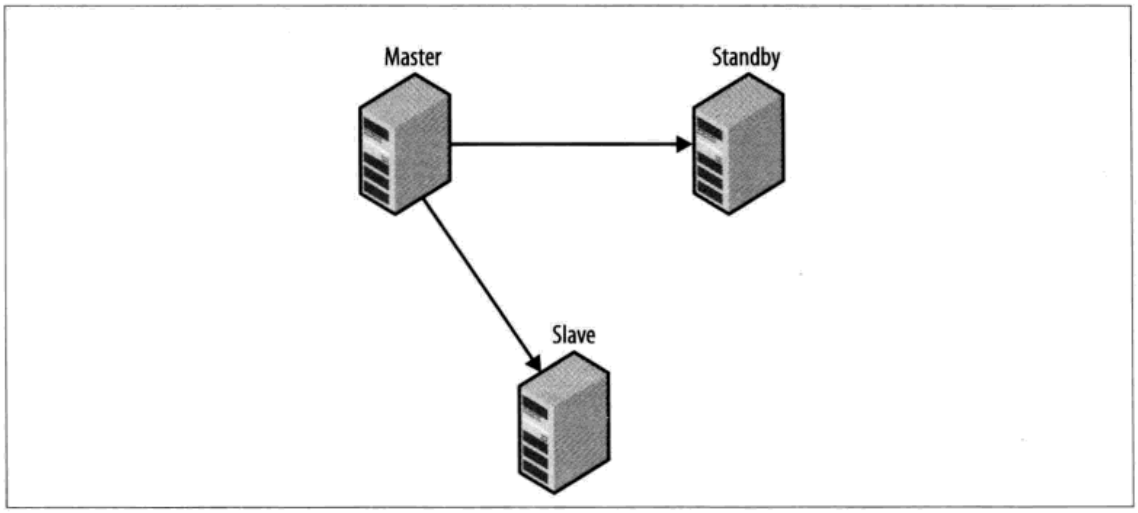


图4-2：热备份服务器和Master

热备份技术原理是指当 Master 发生故障时，热备份提供一个 Master 的原备份，且所有的客户端和 Slave 都可以切换到热备份服务器上并继续工作。正如很多想法一样，现实并不总是那么乐观。

故障是不可避免的，至少在你运行一个大型部署系统时是这样的。服务器是否会挂掉不是问题，问题是什么时候会挂掉，挂掉的频率有多高。如果 Master 因为某些原因出现故障，它不应该使系统终止运行。为了确保操作的继续进行，有必要搭建一个可用的热备份，并在 Master 故障时，把所有的 Slave 都重定向到热备份服务器上，然后你就可以去检查 Master 出现什么问题，或许能修复问题或更换它。在恢复了 Master 后，你可以再让它正常运行，这时可以让它作为一个热备份服务器，或者把所有的 Slave 再重定向到先前的 Master。

听起来很简单，不是么？如果真是那么简单就好了。很不幸，你还需要思考以下这些潜在的问题：

- 当故障发生后，系统转移到热备份 Master 上，你正在从一个新的 Master 上进行复制，所以有必要将 binlog 的位置从 Master 上的位置转换成热备份服务器上的。
- 当 Slave 故障后转移到热备份 Slave，热备份 Slave 可能实际上并没有包含原 Slave 记录的所有更改。
- 当把修复后的 Master 带回配置中后，被修复的 Master 中的二进制日志中有些更改可能从没离开过服务器。

112

所有这些都是相关的问题，但是对于初学者来说，让我们只考虑图 4-3 所示的简单例子。比如，为了对原来的 Master 进行维护，需要执行从正在运行的 Master 到备份服务器的切换。在这种情况下，Master 仍然在运行，所以这种情况变得简单很多，因为我们可以控制 Master，并且让它为我们服务而不会找麻烦。稍后我们再考虑怎样处理因为软件崩溃而导致 Master 发生故障的情况，一个很受挫败的同事可能会去踢服务器，或者门卫被电源线绊倒之类。

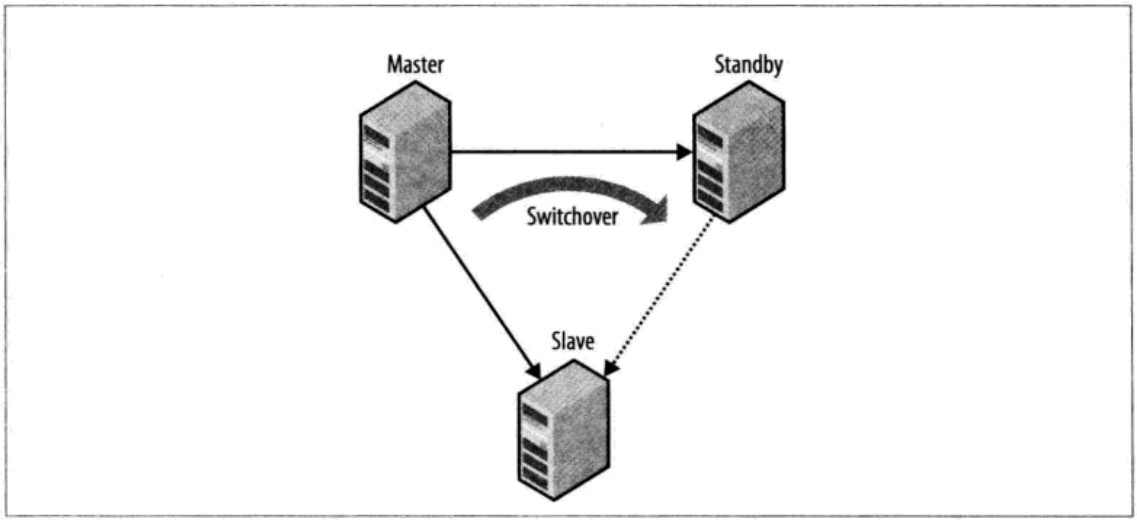


图4-3：从正在运行的Master切换到备份服务器

默认情况下，由 Slave 执行的事件没有被记录到二进制日志中，如果这个 Slave 是 Master 的一个备份，这时会出现问题。在这种情况下，Master 有必要将发送给备份服务器的所有更改写入备份服务器的二进制日志中，如果不这样做，将没有什么可复制的。为了配置备份服务器，在 *my.cnf* 文件中添加一个选项 `log-Slave-updates`。这个选项可以确保来自于 Master 并被执行的语句会被写入 Slave 的二进制日志中。

```
[mysqld]
user                = mysql
pid-file            = /var/run/mysqld/mysqld.pid
socket              = /var/run/mysqld/mysqld.sock
port                = 3306
basedir             = /usr
datadir             = /var/lib/mysql
tmpdir              = /tmp
log-bin             = master-bin
log-bin-index       = master-bin.index
server-id           = 1
lgo-slave-updates
```

113 在更新了选项文件之后，重启服务器。

在这种情况下，切换到备份服务器的主要问题是：进行切换时，在备份服务器上开始复制的地方必须正好是它在 Master 停止复制的地方。如果位置很容易转换，比如，如果 Master 和备份服务器的位置相同，就没这个问题，不幸的是，很多原因导致 Master 和备份服务器的复制位置是不同的。最常见的情况是，当 Master 正在运行时备份服务器没有连到 Master 上，但是即便做到这一点，事件被写入 Master 的二进制文件中的方式也未必与写入备份服务器上的二进制日志中的方式相同。

执行切换时的一个基本思路是：Slave 和备份服务器在完全相同的位置停止运行，然后只要把 Slave 重定向到备份服务器。因为备份服务器在你停止运行后的位置没有任何更改，你只需检查 binlog 的位置即可。然而，仅仅停止 Slave 和备份服务器不一定能保证它们是同步的，所以你必须手动去操作。

为了做到这一点，停止 Slave 和备份服务器时检查双方的 binlog 的位置。因为二者的位置是指向同一台 Master 上的位置，Slave 和备份服务器二者都连接到同一台 Master 上，你可以仅仅根据比较文件名和按照字典顺序的字节位置来检查双方的二进制日志的位置。

```
standby> SHOW SLAVE STATUS\G
     ...
Relay_Master_Log_File: Master-bin.000096
     ...
   Exec_Master_Log_Pos: 756648
1 row in set (0.00 sec)
Slave> SHOW SLAVE STATUS\G
     ...
 Relay_Master_Log_File: Master-bin.000096
     ...
  Exec_Master_Log_Pos: 743456
1 row in set (0.00 sec)
```

在这种情况下，备份服务器是超前 Slave，所以只要写入备份服务器的 Slave 的位置，并启动运行 Slave 直到它赶上备份服务器。为了让 Slave 赶上备份服务器，并在正确的位置停止，使用 START SLAVE UNTIL 命令，就像我们在本章前面描述的停止报告 Slave 时的做法那样。

```
slave> START SLAVE UNTIL
    ->    MASTER_LOG_FILE = 'master-bin.000096',
    ->    MASTER_LOG_POS = 756648;
Query OK, 0 rows affected (0.18 sec)

slave> SELECT MASTER_POS_WAIT('master-bin.000096', 756648);
Query OK, 0 rows affected (1.12 sec)
```

Slave 和备份服务器现在已经精确地停在相同的位置上，一切就绪后，切换到备份服务 114

器，并用 CHANGE MASTER TO 命令切换到备份服务器，把 Slave 连接到备份服务器并重新启动它。但是，你要指定的位置应该在哪儿？因为 Master 在停止运行点记录的文件和位置与备份服务器在同一点记录的文件和位置是完全不同的，有必要去获取当 Master 记录更改时备份服务器记录的位置。为了做到这一点，在备份服务器上运行 SHOW MASTER STATUS 命令。

```
standby> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: standby-bin.000019
        Position: 56447
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

现在你可以用正确的位置，把 Slave 重定向到备份服务器。

```
slave> CHANGE MASTER TO
    ->    MASTER_HOST = 'standby-1',
    ->    MASTER_PORT = 3306,
    ->    MASTER_USER = 'repl_user',
    ->    MASTER_PASSWORD = 'xyzzy',
    ->    MASTER_LOG_FILE = '
standby-bin.000019',
    ->    MASTER_LOG_POS = 56447;
Query OK, 0 rows affected (0.18 sec)

slave> START SLAVE;
Query OK, 0 rows affected (0.25 sec)
```

如果事实是相反的，即 Slave 超前备份服务器，你可以切换之前步骤中的 Slave 和备份服务器的角色。这是可能的，因为 Master 在运行过程中，可以把丢失的更改提供给 Slave 或备份服务器。在下一节中，我们将考虑怎样处理 Master 意外停止的情况，因此不能把丢失的改变提供给 Slave 或备份服务器。

处理Python中的切换

例 4-1 展示了从一个 Slave 切换到另外一个 Master 的 Python 语言的一段代码。replicate_to_position 函数命令一个服务器从 Master 中只读到指定位置。当程序返回时，Slave 将在这个位置停止。switch_to_Master 把 Slave 重定向到一个新的 Master。这个程序假定它执行的服务器和新 Master 都连接到相同的原 Master 上。如果不是这样，位置就没有可比性，程序则引发异常。

115 例4-1：切换到新Master的程序

```
def replicate_to_position(server, pos):
```

```
    server.sql("START SLAVE UNTIL MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s",
    (pos.file, pos.pos))
    server.sql("SELECT MASTER_POS_WAIT(%s,%s)", (pos.file, pos.pos))

def switch_to_master(server, standby):
    stop_slave(server)
    stop_slave(standby)
    server_pos = fetch_slave_position(server)
    standby_pos = fetch_slave_position(standby)
    if server_pos < standby_pos:
        replicate_to_position(server, standby_pos)
    elif server_pos > standby_pos:
         replicate_to_position(standby, server_pos)
    master_pos = fetch_master_position(standby)
    change_master(server, standby, master_pos)
    start_slave(standby)
    start_slave(server)
```

双Master

经常提到的一个高可用性配置是双 Master 拓扑结构。在此设置中，两个 Master 互相复制数据以保持同步。因为是对称的，这种设置使用起来非常简单。发生故障时，切换到备份服务器上不需要重新配置主 Master，当备份服务器发生故障时再切回到 Master 是很容易的。

服务器可以是主动的或者被动的，如果一个服务器是主动的，意味着服务器可以接受写入，就像其他地方使用复制来传播(propagate)一样。如果服务器是被动的，它不接受写入，只是跟随主动的 Master，通常是时刻准备着，等待 Master 出现故障以后代替它。

当使用双 Master 时，有两种不同的配置方法，每种方法的目的不同：

- *Active-active*
 在这个设置中，操作同时写入两个服务器，然后把更改转移到其他的 Master。
- *Active-passive*
 在这个设置中，其中一个 Master 处理写入，被称为主动 Master；而另外一个服务器被称为被动服务器，只是和主动 Master 保持一致。
 这几乎与热备份配置相同，但是因为它是对称的，很容易和 Master 来回切换，轮流成为主动 Master。
 请注意，这个设置并不需要让 Master 响应查询。在本节你能看到的一些解决方法中，被动的 Master 是一个冷备份。

116

这些配置并不一定意味着复制是用于保持服务器同步的，有一些其他技术可以达到这个目的。一些技术可以支持 active-active Masters，而其他一些技术只能支持 active-passive

Masters。

active-active 双 Master 的最常见用途是让服务器从地理位置上接近不同的用户组，比如，在世界的不同地方的办公室。用户可以使用本地服务器，更改会被复制到另外一个服务器以便于两个 Master 都能保持同步。由于事务在本地被提交，系统响应更加灵敏。了解事务在本地提交很重要，这意味着两个 Master 在拥有相同信息的前提下不同步。Master 的更改会被最终传播到另一个 Master，但是在这些都完成之前，两个 Master 上有不一致的数据。

这里有两个你需要注意的主要后果：

- 如果同样的信息在两个 Master 上都被更新，比如，一个用户突然被加入到两个 Master，这两个更新之间将会有冲突，并很可能导致复制停止。
- 当两个 Master 不一致时，如果系统发生崩溃，一些交互将会丢失。

在一定程度上，可以通过只允许写入一个服务器避免更改冲突问题，从而使另一个服务器成为被动的 Master，这被称为主动 - 被动（active-passive）设置，主动的服务器被称为主要的，被动的服务器被称为次要的。

在服务器崩溃时丢失事务是使用异步复制不可避免的结果，但是这取决于应用程序，没有必要把它看成一个严重的问题。当服务器崩溃时，使用 MySQL 5.5 的新功能——半同步复制，它可以限制事务丢失的数量。半同步复制原理是：提交事务的线程会被锁定，直到至少有一个 Slave 收到这个事务。由于事务的事件是在事务被提交到存储引擎后才被发送到 Slave 上的，所以事务的丢失数量可以下降到最多每线程一个。

和主动 - 主动（active-active）方法相似，主动 - 被动（active-passive）设置也是对称的，因此你可以很容易地在 Master 和备份服务器之间来回切换。取决于你处理镜像的方法，也很可能使用被动服务器管理像升级服务器这样的任务，并使用升级服务器作为主动服务器，一旦升级完成，不会有任何停机时间。

117 当使用 active-passive 设置时，有个根本问题需要解决，那就是两台 servers 同时被作为主要 Master 的风险问题，也就是所谓的 *split-brain syndrome*。如果网络在短时间内不能连通，而在这段时间内备份服务器主动提升为主要 Master，但是随后，原来的主 Master 再次成功联机，这时这种风险就可能发生。如果两台服务器都被作为主 Master，此时更改在这两台 Master 上都被执行，这时就有可能发生冲突。当使用共享磁盘时，同时由两个 servers 写入磁盘就有可能导致数据库发生“有趣”的问题，也就是说，可能是危害性的问题并且很难确定。

共享磁盘

一个简单的双 Master 方式如图 4-4 所示，两个 Master 通过共享磁盘的体系结构诸如 SAN（存储区域网络）被连接在一起。在这种方法中，两台服务器被连接到同一个 SAN，并被设置为使用相同的文件。由于其中一个 Master 是被动的，因此当主动的 Master 正常运行时它不会写任何文件，如果 Master 发生故障，这个备用的 Master 将接替它的工作。

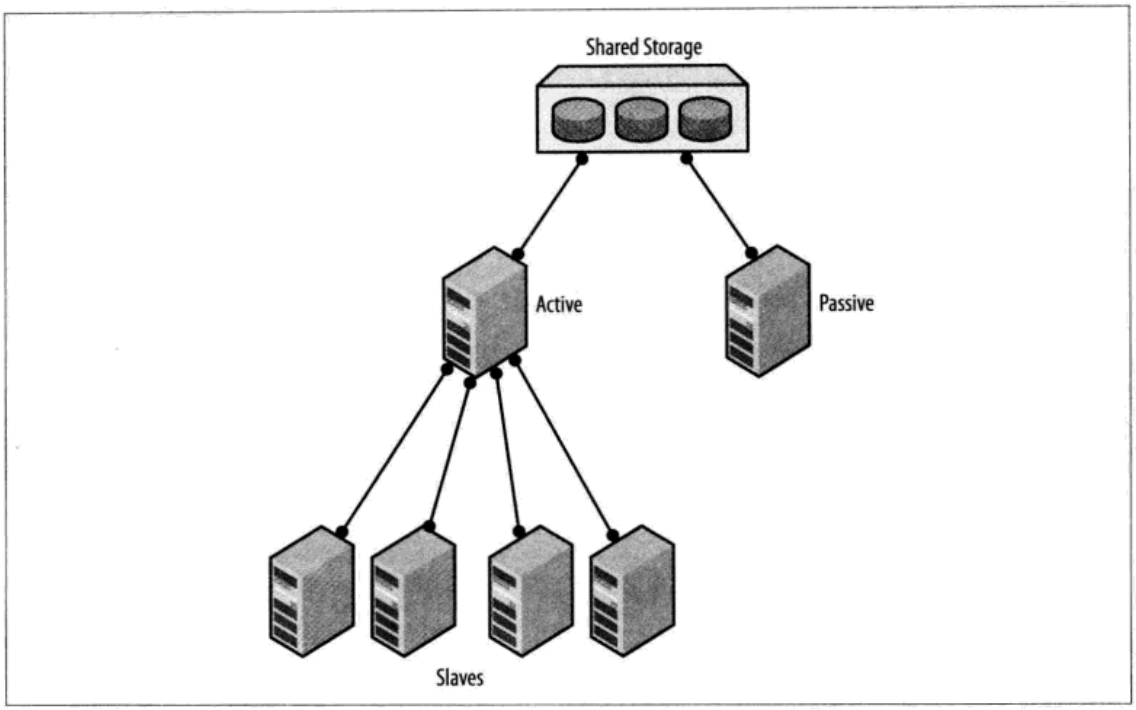


图4-4：使用共享磁盘的双Master

这种方法的优点是，由于 binlog 文件存储在共享磁盘上，所以不需要转换它的 binlog 位置。两台服务器就是彼此真正的镜像，但它们在两个不同的机器上运行。这意味着主 Master 到备份 Master 的切换速度很快，因为 Slave 没有必要将 binlog 位置转换为新 Master 上的 binlog 位置。最有必要的是要注意 Slave 停下来的地方，执行 change Master 命令，并再次开始复制。

118

当你无法使用这个方法时，必须执行表恢复，因为很可能更新被中止。在这种情况下，每个存储引擎的运行都有所不同。例如，InnoDB 必须从事务日志中执行正常恢复，因为它可能位于一个损坏事件中，而如果你使用 MyISAM，可能必须在能继续操作前修复表。InnoDB 是这两种选择中较好的一个，因为其恢复速度比修复 MyISAM 表的速度明显快很多。

例 4-2 显示了使用 Replicant 库解决这个故障的的 Python 脚本。请注意使用 Master 的服务器 ID 的位置，但因为这两个服务器使用相同的文件，备用服务器就是 Master 的镜像。由于此位置也包含服务器 ID，因此这也将获取使用者造成的所有错误，例如进入一个并不是主 Master 镜像的 Master。

例4-2：*在使用共享磁盘时将Slave提升为Master的程序*

```
def remaster_slave(slave, master):
    position = fetch_slave_position(slave)
    change_master(slave, master, position)
```

使用共享磁盘建立双 Masters 的能力取决于使用的共享存储解决方案，但这超出了本书讨论的范围。

使用共享存储的问题是：由于两个 Masters 都使用相同的文件用于存储数据，在被动 Master 上做任何管理任务时必须非常注意。过度写配置文件，哪怕是一点点错误，都可能是灾难性的。

对 split-brain syndrome 的处理依赖于使用的共享磁盘解决方案，这也超出了本书讨论的范围。然而在使用 SCSI 时，发生的一个例子支持通过服务器储存磁盘。这让一台服务器意识到自己真的不再是主 Master，因为它注意到磁盘被存储在另外一台服务器上。

使用DRBD复制磁盘

Linux High Availability project 包含很多对维护高可用性系统有用的工具。这些工具大都超出了本书讨论的范围，但有一个工具对我们而言非常有用：DRBD 技术（分布式复制块设备），这是在网络上复制 block 设备的软件。

图 4-5 显示了一个含有两个 nodes 的典型设置，其中 DRBD 用于复制磁盘到备用服务器。这个设置在每个节点上各创建了一个 DRBD 块设备，DRBD 块设备依次把数据写入到真的磁盘。这两个 DRBD 的进程通过网络连接以确保任何对主 Master 的更改都被复制到
119 备份 Master 上。对于 MySQL 服务器，设备复制是透明的。且 DRBD 设备的外观和行为与正常的磁盘一样，所以在服务器上它不需要做特别的配置。

只能在 active-passive 设置中使用 DRBD 技术，这意味着被动磁盘完全不能被访问。与前面介绍的共享磁盘解决方案和后面描述的双向复制实施方案相比，被动 Master 无法使用，甚至对于单纯的只读任务也是如此。

与共享磁盘解决方案类似，由于两个 DRBD 共享相同的文件，所以它们不需要在两个 Master 间转换二进制日志文件的位置。然而，主 Master 到备用 Master 的切换速度比在前面所述共享磁盘设置的慢。

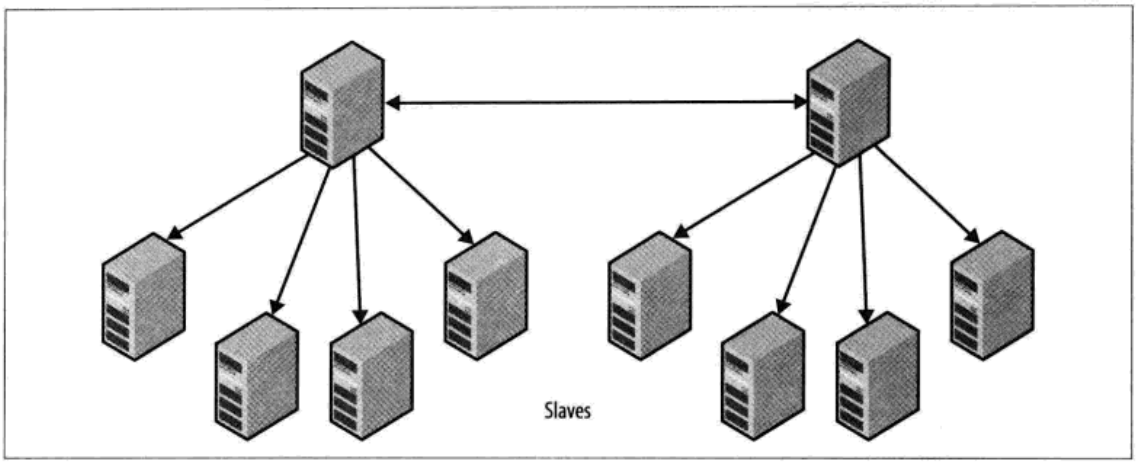


图4-6：双向复制

使用以下步骤设置双向复制：

1. 确保两台服务器有不同的服务器 ID。
2. 确保两个服务器具有相同的数据（并且没有对任何系统做出改变直到复制被激活）。
3. 创建一个复制用户，并准备在两台服务器上复制（使用第 1 章中的知识）。
4. 在两台服务器上同时开始复制。

121



当进行双向复制时，要事先告知复制中没有包括解决冲突的概念。如果两台服务器同时更新一段数据，你将会遇到一个被告知或者没有被告知的冲突。如果幸运，复制将会停止在有冲突的语句处，但是不要指望它一定会出现。如果你打算创建一个高可用性的系统，请确保在应用程序级别不要让两台服务器同时更新一个数据。

即使数据是自然分段的（如前面提到过的例子：两个办公室分布在不同的地点），也要规定以确保数据不要在错误的服务器上被偶尔更新，这是至关重要的。

在这种情况下，应用程序必须要连接到员工负责的服务器上进行信息更新，而不仅仅是在本地进行数据更新，并期望有最好的结果。

如果你想要把 Slave 连接到其中一个服务器，应该确保 `log-Slave-updates` 选项是开启的。由于其他 Master 同样会作为 Slave 连接到这里，一个显而易见的问题是：当被服务器发送出去的事件返回到服务器时，这些事件会发生什么？

当复制在进行时，产生这个事件的服务器的 ID 会附加到每一个事件里。当 Slave 把事件写到二进制日志时，这个服务器 ID 会遍布更广。当服务器看到一个和它自己的服务器 ID 同样的事件 ID 时，就会自然地跳过这个事件，复制会在下一个事件继续进行。

有时候，你无论如何都想要处理这个事件。像下面的情况，如果已经移除了旧的服务器并且创建了一个有同样 ID 的新服务器，你是在履行一个 PITR 过程。在那些情况下，使用 replicate-same-server-id 配置变量禁止此检查是可能的。但是不可以在 log-Slave-updates 设置开启的同时来操作这个选项。否则，发送事件极可能进入一个循环并且快速击溃所有的服务器。为了阻止这种情况发生，在使用 replicate-same-server-id 的时候，不允许进行其他事件。

当使用一个 active-active 设置时，有必要安全地处理冲突，目前为止最简单的方法（也是目前处理 active-active 设置最值得推荐的方法）是保证不同的活动服务器写到不同的区域。

一个可能的方案是分配不同的数据库（或者不同的表）到不同的 Master 里。例 4-3 展示了一个这样的设置安装：用两个不同的表，每一个表都被不同的 Master 更新。为了更简便地看到分割的数据，创建了合并两个表的视图。

例 4-3：对不同的办公室使用不同的表

```
CREATE TABLE Employee_Sweden (
   uid INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(20)
);

   CREATE TABLE Employee_USA (
   uid INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(20)
    );

-- 这个视图在同时从两个表读取的时候使用
CREATE VIEW Employee AS
    SELECT 'Swe', uid, name FROM Employee_Sweden
  UNION
    SELECT 'USA', uid, name FROM Employee_USA;
```

122

如果分割是自然的，这个方法是最好用的，例如，不同的办公室对它们本地的数据有不同的表，并且数据只需在上报时合并。看起来够简单，但是下面的问题可以使表的使用和管理变得复杂：

- 读写到不同的表

 因为这样的视图是定义好的，所以不能更新它。必须要直接写入到真正的表中，然而可以选择读取视图或者直接从表读取。

 因此，使用应用程序逻辑来处理读取和写入到不同的表的划分也许是有必要的。
- 准确数据和当前数据

 由于不同的表被不同的站点管理，因此同时更新到两个表将会导致系统暂时进入这

样的状态：两个服务器都有在其他的服务器上不可用的信息。如果此时对信息做快照，这个数据将会不准确。

如果需要准确的信息，那么需要确保信息的产生方法是准确的。由于这种方法是高度依赖于应用程序的，这里不过多介绍。

- 优化的视图

 当使用视图时，有两种技术可用于建立一个结果集。在第一个方法中（叫做MERGE），这个视图可以扩展到适当的位置，可以被优化，并且可以被当做类似于SELECT查询一样执行。在第二个方法中（叫做TEMPTABLE），会构造一个填充了数据的临时表。

 如果服务器使用TEMPTABLE视图，它会运行得很糟糕，而MERGE 视图类似于对应的SELECT。MySQL 使用TEMPTABLE，任何时候，视图的定义都没有指出视图的行和基础表中的行有简单的一一映射（例如，如果视图定义包括UNION，GROUP BY，子查询或聚合功能）因此，对视图做仔细的设计是获得良好性能要素。

 在这两种情况下，必须考虑使用视图报告的影响，因为这可能会影响性能。

123 如果给每一个服务器分配单独的表，不会发生任何冲突，因为更新完全被分开了。然而，如果所有的站点必须去更新同一个表，就得考虑其他的方案了。

MySQL 服务器有处理这种情况的方案，体现在下面两个变量上：

- auto_increment_offset

 这个变量控制着表中任何AUTO_INCREMENT列的起始值。这个值是第一行插入到表中时从AUTO_INCREMENT列得到的值。对于接下来的各行，每个值是 使用auto_increment_increment 来计算的。

- auto_increment_increment

 这是一个用于计算一个AUTO_INCREMENT列下一个值的增量。



这两个变量有会话和全局版本，它们影响服务器上的所有表，而不是仅仅影响被创建的表。每当新行插入到表的AUTO_INCREMENT列中时，接下来各行的值将会应用到下面的公式计算：

$$value_N = auto_increment_offset + N*auto_increment_increment$$

注意：下一个数值并不是通过给表中最后一个值加auto_increment_increment计算出来的。

用auto_increment_offset和auto_increment_increment来确保表中添加的新行被分配不同的数字序列中的数字，这取决于哪个服务器在被使用。比如，第一个服务器用序

列 1，3，5…（奇数）时，第二个服务器在用序列 2，4，6…（偶数）。

例 4-3 和例 4-4 中使用这两个变量确保插入新员工到员工表中时，两个服务器用的是不同的 ID。

例4-4：两个服务器同时对同一个表进行写操作

```
-- 这个公共的表可以在任一个服务器上创建
CREATE TABLE Employee (
   uid INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(20),
   office VARCHAR(20)
);
-- 设置第一个 Master
SET GLOBAL AUTO_INCREMENT_INCREMENT = 2;
SET GLOBAL AUTO_INCREMENT_OFFSET = 1;

-- 设置第二个 Master
SET GLOBAL AUTO_INCREMENT_INCREMENT = 2;
SET GLOBAL AUTO_INCREMENT_OFFSET = 2;
```

这个方案处理了这个表中的新条目的插入，但是当数据被更新的时候，确保更新的指令被发送到了正确的服务器依然是至关重要的（负责员工的服务器）。否则，数据可能是不一致的。如果更新没有被正确地处理，Slave 正常情况下不会停止，它们只会继续复制信息，这将导致两个服务器中的数据不一致。 124

例如，如果第一个 Master 执行了下面的语句：

```
master-1> UPDATE Employee SET office = 'Vancouver' WHERE uid = 3;
Query OK, 1 rows affected (0.00 sec)
```

同时，用下面的语句将同一行更新到第二个服务器上：

```
master-2> UPDATE Employee SET office = 'Paris' WHERE uid = 3;
Query OK, 1 rows affected (0.00 sec)
```

结果是第一个 Master 将会放置巴黎的员工信息，而第二个 Master 放置温哥华的员工信息（注意：顺序将会调换，因为每一个服务器会更新它之后的其他服务器的语句）。

发现并且阻止这种不一致行为是很重要的，因为随着时间的推移，它们会传播并创造出更多的不一致性。基于语句的复制执行基于两个服务器上的数据的语句，因此一个不一致会导致更多的不一致发生。

如果你小心地分开前面提到的两个服务器所做的更改，该行的更改将被复制并且两个 Master 会因此保持一致。

如果用户在不同的服务器上用不同的表，最简便的方法是阻止此类错误分配权限，这样

一个用户就不能偶然地在错误的服务器上更改表。但是，这并非总是可能的，而且也不能防止刚刚所描述的情况。

半同步复制

Google 为 MySQL 和 InnoDB 设计了一个大规模补丁集以量身打造服务器和存储引擎。其中一个修补程序可用于 MySQL 5.0 版本，是半同步的复制补丁。MySQL 已经打上了该补丁并在 MySQL 5.5 中发布了。

半同步复制的理念是在允许更改操作继续执行前，确保更改操作至少被写入一个 Slave 的磁盘。这意味着对于每一个连接，最多只有一个事务会由于 Master 崩溃而丢失。

重要的是要明白半同步复制补丁没有暂停提交事务，它只是在事务已被写入到至少一个 Slave 的中继日志中之前，避免发送一个答复给客户端。图 4-7 显示了在提交事务时发出的指令顺序。如你所见，在事务发送到 Slave 之前，它被提交到存储引擎，但只有当 Slave 被告知事务已经在持久存储中之后，客户端的提交指令才会返回。

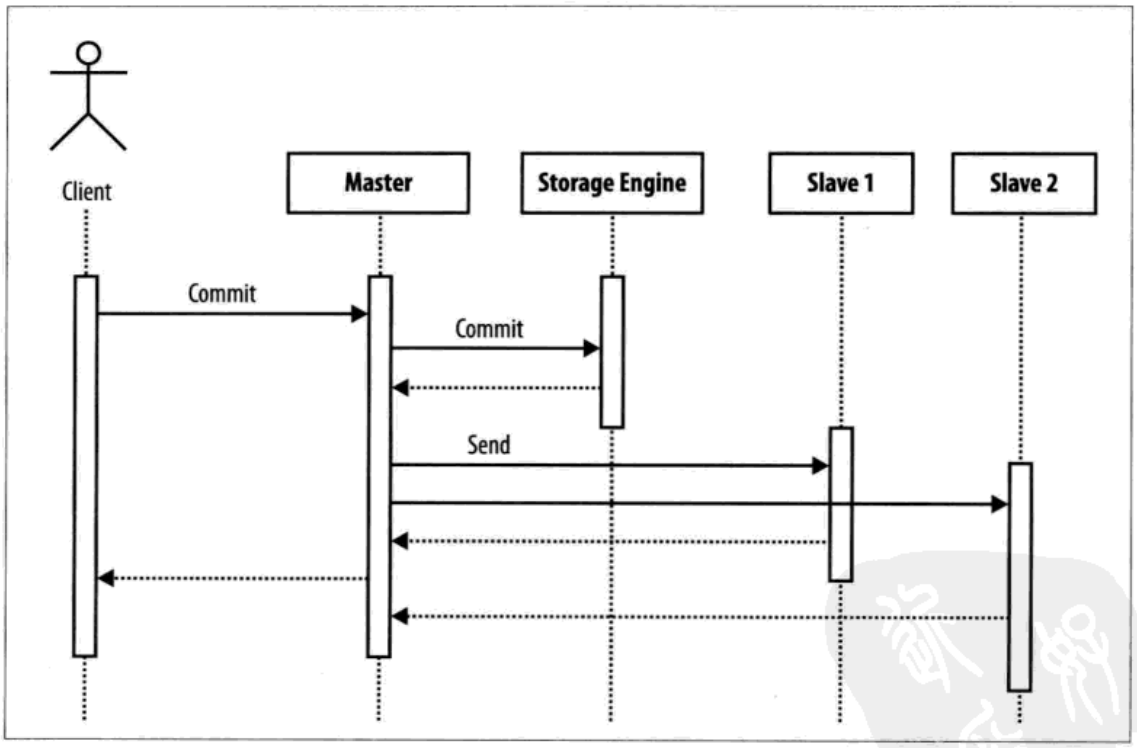


图4-7：半同步复制中的事务提交过程

125 这意味着，在事务被提交给存储引擎之后但还没提交给 Slave 之前，如果发生系统崩溃，则每个连接都有可能丢失一个事务。然而，由于事务是在已被提交到的 Slave 后再被确

认已提交给客户端的，因此最多只会丢失一个事务。

这通常意味着，每个客户端最多有一个事务丢失，但如果客户端同时和 Master 有多个连接，这时客户端同时提交多个事务，并且服务器崩溃，那么每个连接就会丢失一个事务。

配置半同步复制

要使用半同步复制，Master 和 Slave 都需要能支持它，所以无论是 Master 和 Slave 必须运行 MySQL 5.5 或更高版本，并且启用半同步复制机制。如果 Master 或 Slave 不支持半同步复制，它不会被使用，但复制可以正常工作，通常这意味着多个事务可能丢失，除非有特殊的预防措施可以确保新的事务开始前每一个事务都能到达 Slave。

使用以下步骤启用半同步复制：

126

1. 在 Master 上安装 Master 插件：

```
master> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
```

2. 在每台 Slave 上安装 Slave 插件：

```
slave> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
```

3. 一旦你已经安装了这些插件，在 Master 和 Slave 上启用这些插件。这是通过两个服务器变量控制的，同时作为选项可供选择。为了确保即使重新启动，设置也能生效，最好减少服务器，并且把选项添加到 Master 的 *my.cnf* 文件中：

```
[mysqld]
rpl-semi-sync-master-enabled = 1
```

从 Slave 中启用：

```
[mysqld]
rpl-semi-sync-slave-enabled = 1
```

4. 重启服务器。

如果按照刚才的指示做了，现在就有了一个半同步复制 setup，并可以对它进行测试，但是需要考虑以下情况。

- 如果所有的 Slave 崩溃怎么办？如果你只有一个服务器（这不是不可能的），从而没有一个 Slave 确认它已将事务存储到中继日志中，这时会发生什么？
- 如果所有的 Slave 断开怎么办？在这种情况下，没有 Slave 可供 Master 出于安全保护而发送事务。

除了 rpl-semi-sync-master-enabled 和 rpl-semi-sync-slave-enabled，还有两个选项可以处理以上情况：

- rpl-semi-sync-master-timeout=*milliseconds*
 为了防止半同步复制在没有收到确认的情况下发生堵塞，可以使用 rpl-semi-sync-mastertimeout=*milliseconds* 选项设置一个计时器。
 如果 Master 在超时之前没有收到任何确认，将恢复到正常异步复制，并继续执行没有半同步复制的操作。
 此选项可作为服务器变量，并在没有减少服务器的情况下被设置。但是请注意，作为每一个服务器变量，这个值在重启服务器后将不被保存。
- rpl-semi-sync-master-wait-no-slave={*ON*|*OFF*}
 如果一个事务被提交，但 Master 没有任何 Slave 连接，这时 Master 不可能将事务发送到其他地方保护起来。默认情况下，Master 会在时间限制范围内继续等待 Slave 的连接，并确认该事务已被正确写入到磁盘上。
 127
 可以使用 rpl-semi-sync-master-wait-no-slave={*ON*|*OFF*} 选项关闭这种行为，在这种情况下，如果没有 Slave 连接，Master 会恢复到异步复制。

监控半同步复制

这两个插件都安装了大量的状态变量，可以利用这些变量来监控半同步复制。这里将介绍最有用的一些变量。要获取完整的列表信息，请查阅半同步复制在线参考手册（*http://dev.mysql.com/doc/refman/5.5/en/replication-semisync-interface.html*）。

- rpl_semi_sync_master_clients
 此状态变量报告了支持和注册半同步复制的已连接的 Slave 数量。
- rpl_semi_sync_master_status
 Master 的半同步复制状态，1 是活动状态，0 是非活动状态——要么是因为它没有被启用，或是因为它已恢复到异步复制。
- rpl_semi_sync_slave_status
 Slave 上的半同步复制状态，如果是 1，也就是说它已被启用，且 I / O 线程正在运行，如果是 0，就是处于非活动状态。

可以使用 SHOW STATUS 命令或通过信息模式表 GLOBAL_STATUS 来阅读这些变量。如果一样把这些值用于其他用途，SHOW STATUS 命令是很难使用的，且正如示例 4-5 中显示的查询，它使用信息模式中的 SELECT 提取信息并将这个信息存储在一个用户定义的变量中。

例4-5：使用信息架构检索值

```
master>SELECT Variable_value INTO @value
    -> FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    -> WHERE Variable_name  = 'Rpl_semi_sync_master_status';
Query OK, 1 row affected (0.00 sec)
```

Slave的提升

如果你有一个可运行的 Master，并且可以在切换 Master 之前，使用 Master 同步备份服务器和 Slave，这时程序可以很好地运行，但当 Master 突然死机时会发生什么？由于复制已在所有的 Slave（包括备用）上停止，它将无法运行复制，充其量只能获取所有必要的更改，并将与新 Master 同步。

如果备用服务器超前于所有需要重新分配的 Slave，这是没问题的，因为每个 Slave 都可以从备用服务器停止复制的地方开始复制。

你将丢失所有在 Master 上执行过的、但尚未被发送到备用服务器的更改。我们将单独讨论在这种情况下如何处理 Master 的恢复。 128

如果备用服务器滞后于 Slave 中的一个，不应该使用备用服务器作为新的 Master，因为 Slave 比备用服务器的数据更多，事实上，使用“数据更多的”Slave 替换 Master 将更好，因为 Slave 复制了原 Master 的大部分事件。

这正是使用提升 Slave 来处理 Master 故障的方法：不是试图保持一个专门的备用，而是确保任何一个连接到 Master 的 Slave 能够在 Master 发生故障的时候被提升为 Master。通过选择“最可信任的”的 Slave 作为新的 Master，确保其他的 Slave 都不会比新 Master 更可信，这样它们就可以连接到新的 Master，并从新 Master 上读取事件。

然而，还有一个关键的问题需要解决——Slave 与新的 Master 的同步问题，以确保没有任何事件丢失或重复。在这种情况下出现的问题是：所有的 Slave 都需要读取来自于新 Master 的事件，但是新 Master 和老 Master 的位置不同。那么，一个不专业的数据库管理员能够做什么？

提升Slave的传统方法

在得出最终解决方案之前，让我们先来看看处理 Slave 提升的推荐做法，这个方法将可以很好地介绍这个问题，也使我们能够精确定位需要处理棘手问题的最终解决方案。

图 4-8 显示了一个典型的 Master 和多个 Slave 的格局。

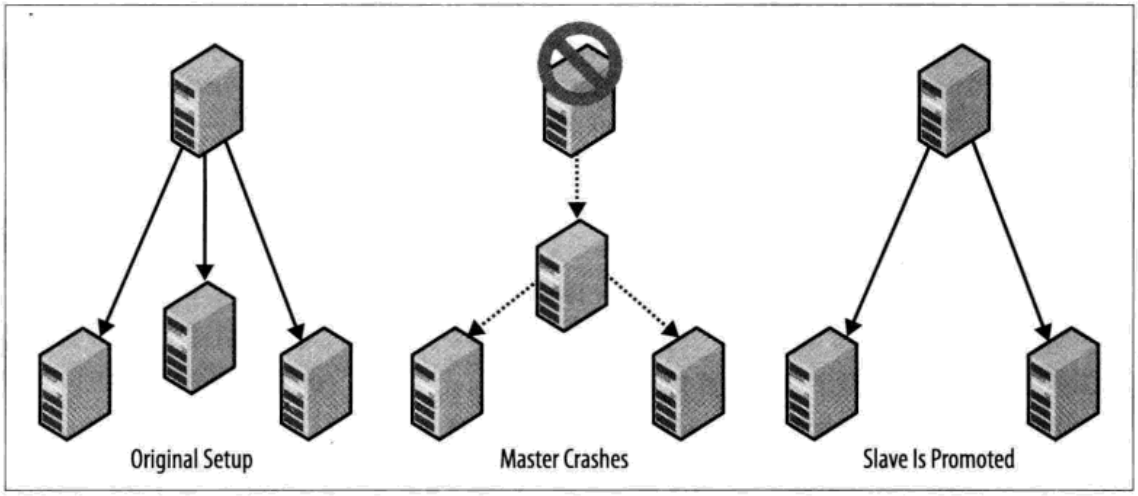


图4-8：升级Slave取代出现故障的Master

129 在传统的 Slave 提升方式中，必须实施下列各项：

- 每个可提升的 Slave 必须有一个复制用户的账户。
- 每个可提升的 Slave 必须在运行时启用 `--log-bin` 选项，也就是说，需要激活二进制日志。
- 每个可提升的服务器必须在运行时不启用 `--log-slave-updates` 选项（其原因将会在短期内变得显而易见）。

假设你启用了如图 4-8 所示的原始安装系统，而 Master 发生故障，可以通过如下步骤将一个 Slave 提升为新的 Master：

1. 使用 `STOP SLAVE` 停止 Slave。
2. 使用 `RESET MASTER` 重新设置即将成为新 Master 的 Slave。这将确保 Slave 作为一个新 Master 开始被启用，而任何连接的 Slave 将开始从新 Master 中读取事件。
3. 使用 `CHANGE MASTER TO` 将其他的 Slave 连接到新的 Master 上。由于重新设置了新的 Master，可以从二进制日志的起点开始复制，因此没有必要给 `CHANGE MASTER TO` 提供任何位置信息。

不幸的是，这个方法是基于一个通常不真实的假设——Slave 已经接收到 Master 上产生的所有改变。在一个典型的格局中，Slave 将在不同程度上落后于 Master。可能只是少数事务滞后，但尽管如此，它们还是落后了。下一章将介绍该问题的解决方法。

无论如何，这种方法非常简单，如果你能处理丢失的事务或者你是在低负荷下运行的，那么这是有用的。

Slave提升的修订方法

Slave 提升的传统方法在大多数情况下是不够用的，因为 Slave 往往落后于 Master。图 4-9 说明了当 Master 出乎意料地消失时的典型情况。中间标有“二进制日志”的框是 Master 的二进制日志，每个箭头代表 Slave 执行了多少二进制日志。

在这个图中，每个 Slave 都停止在不同的 binlog 位置。为解决这个问题并将系统重新联机，一个 Slave 必须被选为新的 Master（最好是有最近 binlog 位置的那个）且其他的 Slave 都必须与新 Master 同步。

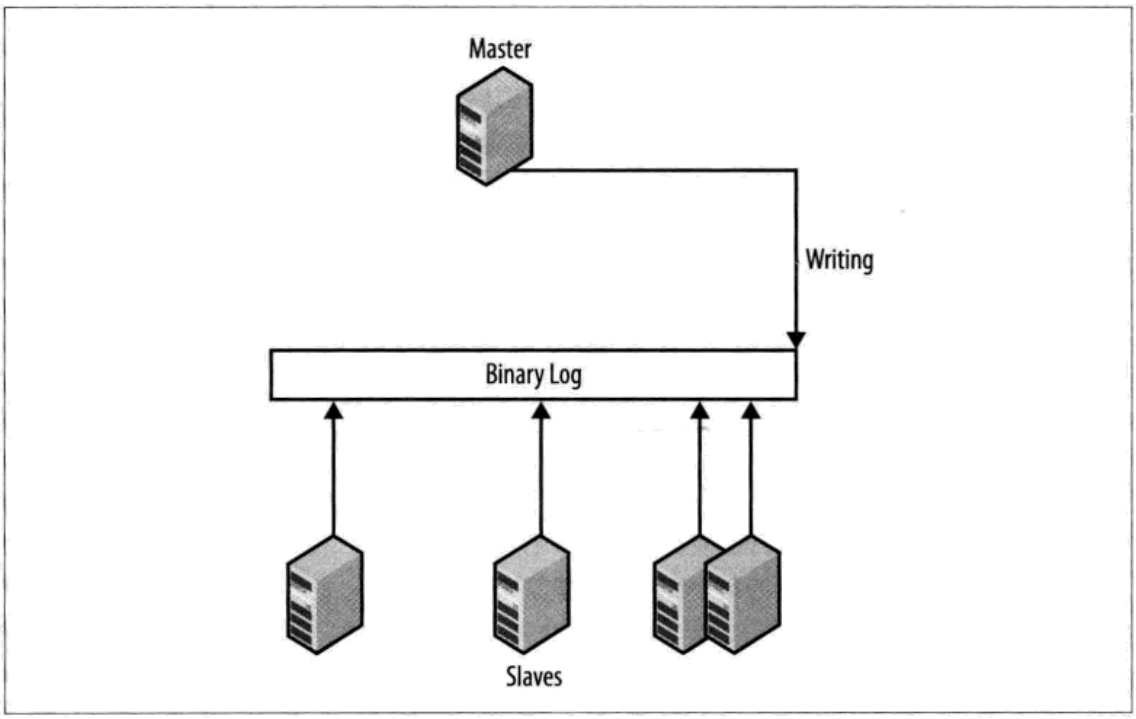


图4-9：Master和连接的Slave的二进制日志位置

关键问题在于将每个 Slave 的位置转换（在已宕机 Master 中的位置）到被提升的 Slave 上的位置。遗憾的是，已经执行的事件的历史记录及它们对应在 Slave 上的 binlog 位置在复制过程中丢失了——每次 Slave 执行来自于 Master 的事件，它都写一个新事件到二进制日志中，伴随一个新的 binlog 位置。对于同样的事件，Slave 的位置与 Master 的二进制位置毫无关系。对于我们来说唯一的选择仍然是扫描被提升的 Slave 的二进制日志。 130
使用下面的技术：

- 启用二进制日志；否则，没有更改可以被复制。
- 启用记录 Slave 的更新（使用 `log-slave-updates` 选项）；否则，从原来的 Master 上来的更改不会被转送。

- 每个 Slave 需要有一个复制用户来担当 Master，因此如果它成为新 Master 的最佳候选，其他 Slave 可以连接到它，并从中复制事件。

对没有被提升的每个 Slave 执行以下步骤：

1. 弄明白它执行的最后一个事务。
2. 找出被提升的 Slave 的二进制日志中的事务。
3. 从被提升的 Slave 上取得事务的 binlog 位置。
4. 未被提升的 Slave 从被提升的 Slave 上的位置开始复制。

为了将每个 Slave 上的最新的事务与被提升的 Slave 的二进制日志中的事件相对应，需要为每个事务加标签。标签的内容和结构并不重要，无论谁实行该事务，它都需要被唯一标识，因此 Master 上的每个事务都可以在被提升的 Slave 的二进制日志中找到。这种类型的标签，称为 *global transaction ID*。

最简单的实现方式是在每个事务的结尾插入一条语句去更新一个特殊的表，并用它来追踪每个 Slave 在哪里。只需在每个事务提交之前，有一个语句使用唯一标识事件的数字更新表信息。

131

标签的处理主要有两种方式：

- 扩展应用程序代码来执行必要的语句。
- 调用一个存储过程来执行每个提交并在程序中写标签。

由于第一种方式更容易被接受，这里将演示它。如果你对第二种方式感兴趣，请看后面的“提交事务的存储过程”。

为执行 global transaction ID，我们已经在例 4-6 中创建了两张表：一张表名叫 *Global_Trans_ID* 用来生成序列号，而另一个表名叫 *Last_Exec_Trans* 用来记录 global transaction ID。

server ID 被加入 *Last_Exec_Trans* 的定义中，用来区分在不同服务器上提交的事务。例如，如果在所有 Slave 设法连接上之前，被提升的 Slave 发生故障，这时区分原始 Master 的事务 ID 和被提升的 Slave 的事务 ID 是很重要的。否则，当重定向到第二个被提升的 Slave 时，没有设法连接上被提升的 Slave 的 Slave 将从一个错误的位置开始执行。这个例子使用 MyISAM 来定义计数器表，但用 InnoDB 也可以做到。

例4-6：用来生成和跟踪global transaction ID的表

```
CREATE TABLE Global_Trans_ID (
```

```
    number INT UNSIGNED AUTO_INCREMENT PRIMARY KEY
) ENGINE = MyISAM;

CREATE TABLE Last_Exec_Trans (
    server_id INT UNSIGNED,
    trans_id INT UNSIGNED
) ENGINE = InnoDB;

-- Insert a single row with NULLs to be updated.
INSERT INTO Last_Exec_Trans() VALUES ();
```

下一步是建立一个程序来将 global transaction ID 添加到二进制日志中，因此提升 Slave 的程序可以从日志中读取 ID。下面的程序适用于我们的目的。

1. 在事务计数表中插入一条数据，并确保在这之前关闭二进制日志，因为插入不会被复制到 Slave 上。

```
master> SET SQL_LOG_BIN = 0;
Query OK, 0 rows affected (0.00 sec)
master> INSERT INTO Global_Trans_ID() VALUES ();
Query OK, 1 row affected (0.00 sec)
```

132

2. 使用函数 LAST_INSERT_ID 来获得 global transaction ID。为简化这个逻辑，同时从服务器变量 server_id 获取 server ID。

```
master> SELECT @@server_id as server_id, LAST_INSERT_ID() as trans_id;
+-----------+----------+
| server_id | trans_id |
+-----------+----------+
|         0 |      235 |
+-----------+----------+
1 row in set (0.00 sec)
```

3. 在插入 global transaction ID 到 *Last_Exec_Trans* 跟踪表之前，可以从计数表删除这一行以节约空间。这个可选步骤只适用于 MyISAM 表。如果你使用 InnoDB，必须小心把最后使用的 global transaction ID 留在表中。InnoDB 是根据表中当前存在的自增字段的最大数值来确定下一个数字的。

```
master> DELETE FROM Global_Trans_ID WHERE number < 235;
Query OK, 1 row affected (0.00 sec)
```

4. 打开二进制日志

```
master> SET SQL_LOG_BIN = 1;
Query OK, 0 rows affected (0.00 sec)
```

5. 用你在第 2 步得到的 server ID 和 transaction ID 去更新 *Last_Exec_Trans* 追踪表。这是通过 COMMIT 提交事务前的最后一个步骤。

```
master> UPDATE Last_Exec_Trans SET server_id = 0, trans_id = 235;
Query OK, 1 row affected (0.00 sec)
master> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

每个 global transaction ID 代表复制可以重新开始的那个点。因此，你必须为每个事务执行这个程序。如果某个事务不使用它，该事务将不会被正确地标上标签，也将不可能从这个位置开始复制。

现在，为了在 Master 发生故障后将一个 Slave 提升为新 Master，必须从所有 Slave 中找到那个有最近更改的 Slave（也就是说，有最大的 binlog 位置）并将这个 Slave 提升为 Master。然后让每个其他的 Slave 连接到它。

为了连接到被提升的 Slave 的 Slave 在正确的位置开始复制，必须找到被提升的 Slave 上最后执行的事务的位置。扫描被提升的 Slave 上的二进制日志，从而找到正确的 transaction ID。

133 使用下面的步骤来执行恢复：

1. 停止 Slave。从它的 *Last_Exec_Trans* 表中获得最后 global transaction ID。
2. 选取有最高的 global transaction ID 的 Slave，将其升级为 Master。如果有多个，挑选一个。
3. 使用 `SHOW MASTER LOGS`，得到将要被提升的的 Slave 的 Master 位置，同时得到 Slave 的二进制日志。请注意，`SHOW MASTER LOGS` 的最后一行与你在 `SHOW MASTER STATUS` 中看到的是相当的。
4. 使被提升的 Slave 联机，并让其开始接受更新。
5. 连接到被提升的 Slave，然后扫描二进制日志，找到你在每个 Slave 的二进制日志中找到的最新的 global transaction ID。除非你找到一个已知是合适的文件位置，否则对于读取二进制日志来说，唯一合适的开始位置就是起点。因此，你必须从最后一个开始以反序扫描二进制日志。

 这个步骤将为你在步骤 1 中收集的每个 global transaction ID 提供被提升的 Slave 上的二进制日志位置。
6. 重新连接每个 Slave 到被提升的 Slave，在 Slave 上为恢复所有信息而需要开始的位置开始复制，并使用步骤 5 的信息。

前面的 4 步是简单的，而第 5 步则有点复杂。为说明情况，让我们看一个例子。这个例

子从前面三步收集基本信息。表 4-2 列出三个带有 global transaction ID 的样本 Slave。

表4-2：所有连接上的Slave的Global transaction ID

	Server ID	Trans ID
slave-1	1	245
slave-2	1	248
slave-3	1	256

如表 4-2 所示，Slave-3 有最后的 Global transaction ID，因此有必要将每个 Slave 的 global transaction ID 转换为 Slave-3 的二进制日志位置，这样，我们就需要 Slave-3 上的二进制日志的信息，这些信息我们将在例 4-7 中得到。

例4-7：即将被提升的Slave-3的Master位置

134

```
slave-3> SHOW MASTER LOGS;
+--------------------+-----------+
| Log_name           | File_size |
+--------------------+-----------+
| slave-3-bin.000001 |      3155 |
| slave-3-bin.000002 |    345217 |
| slave-3-bin.000003 |     24665 |
| slave-3-bin.000004 |    788243 |
| slave-3-bin.000005 |      1778 |
+--------------------+-----------+
5 row in set (0.00 sec)
```

从 SHOW MASTER LOGS 的输出中所要知道的重要的信息就是日志的名字，因此你可以为 global transaction ID 而扫描它们。例如，当用 mysqlbinlog 读取 *Slave-3-bin.000005* 文件时，部分输出如例 4-8 所示。Slave-3 收到的从位置 596 开始的事务（在输出的第一行中高亮显示）有 Slave-1 接收的 global transaction ID，正如对 Last_Exec_Trans 表的一个 UPDATE 所示。

例4-8：对于一个事务通过mysqlbinlog命令得到的的输出

```
# at 596
#091018 18:35:42 server id 1 end_log_pos 664 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
BEGIN
/*!*/;
# at 664
#091018 18:35:42 server id 1 end_log_pos 779 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
UPDATE user SET messages = messages + 1 WHERE id = 1
/*!*/;
# at 779
#091018 18:35:42 server id 1 end_log_pos 904 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
```

```
INSERT INTO message VALUES (1,'MySQL Python Replicant rules!')
/*!*/;
# at 904
#091018 18:35:42 server id 1 end_log_pos 1021 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
UPDATE Last_Exec_Trans SET server_id = 1, trans_id = 245
/*!*/;
# at 1021
#091018 1908:35:42 server id 1 end_log_pos 1048 Xid = 1433
COMMIT/*!*/;
```

表 4-2 显示了 trans_id 245 是 Slave-1 的最后一个事务，因此现在你知道 Slave-1 的开始位置是在文件 *Slave-3-bin.000005* 的 1048 字节位置。因此为在正确的位置启动 Slave- 1，现在可以执行 CHANGE MASTER TO 和 START SLAVE:

```
slave-1> CHANGE MASTER TO
      ->    MASTER_HOST = 'slave-3',
135>  ->    MASTER_LOG_FILE = 'slave-3-bin.000005',
      ->    MASTER_LOG_POS = 1048;
Query OK, 0 rows affected (0.04 sec)

slave-1> START SLAVE;
Query OK, 0 rows affected (0.17 sec)
```

通过这种方式追溯（定位在程序中你第一步记录的每个事务）你可以一个接一个地连接 Slave 到新的 Master 的正确位置。

如果 update 语句被添加进每个事务的提交，该技术将会很有效。遗憾的是，在语句提交之前和之后都有一些语句执行了隐式的提交。典型的例子包括 CREATE TABLE, DROP TABLE, 和 ALTER TABLE。由于这些语句做了隐式的提交，它们不能被正确地标签，因此不可能恰好在他们之后重启。这意味着，如果例 4-9 中的一系列语句被执行而同时发生系统崩溃，将可能发生潜在问题。

如果 Slave 只执行了 CREATE TABLE 接着丢失了 Master，最后看到的 global transaction ID 就为 INSERT INTO 的，也就是说，就在 CREATE TABLE 语句之前。因此，Slave 将试图用 INSERT INTO 语句的 transaction ID 重新连接到被提升的 Slave。由于它将找到其在被提升的 Slave 的二进制日志中的位置，因此它将从再次复制 CREATE TABLE 语句开始，从而导致 Slave 因为一个错误而停止。

可以通过小心地使用和设计语句来避免这些问题；例如，如果 CREATE TABLE 被 CREATE TABLE IF NOT EXISTS 语句所取代，Slave 将注意到该表已经存在而跳过执行该语句。

例4-9：global transaction ID不能被指派的语句

```
INSERT INTO message_board VALUES ('mats@sun.com', 'Hello World!');
CREATE TABLE admin_table (a INT UNSIGNED);
INSERT INTO message_board VALUES ('', '');
```

Python中的Slave提升

你现在已经看到 Slave 提升的两种技术：一个传统的技术会因在一些 Slave 上丢失事务而受损，而一个更复杂的技术是可以恢复所有可用的事务。传统的方法用 Python 很容易实现，因此我们关注更复杂的那一个。为处理这个方式的 Slave 提升，有必要：

- 正确地配置所有的 Slave
- 添加表 *Global_Trans_ID* 和 *Last_Exec_Trans* 到 Master。
- 提供应用程序代码以正确地提交事务。
- 写代码来自动执行 Slave 的升级。

可以使用 `Promotable` 类（如例 4-10 所示）来处理新类型的服务器。如你所见，这 136
个例子再次使用了前面介绍的 `_enable_binlog` 辅助方法，并增加了一个方法来设置 `log-Slave-updates` 选项。由于可升级的 Slave 需要我们在前面为 Master 所展示的那些特殊的表，可升级的 Slave 额外还需要添加到 *Master* 上的表。要做到这一点，我们写了个名叫 `_add_global_id_tables` 的函数。这个函数假设如果这些表已经存在，它们有正确的定义，就不试图去重新创建它们。然而，*Last_Exec_Trans* 表需要从 update 的那一行开始正确工作，因此如果没有警告显示表已经存在，那么我们创建这个表，然后添加一行 NULL 到这个表中。

例4-10：可升级的Slave角色的定义

```
_GLOBAL_TRANS_ID_DEF = """
CREATE TABLE IF NOT EXISTS Global_Trans_ID (
  number INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (number)
) ENGINE=MyISAM
"""

_LAST_EXEC_TRANS_DEF = """
CREATE TABLE IF NOT EXISTS Last_Exec_Trans (
  server_id INT UNSIGNED DEFAULT NULL,
  trans_id INT UNSIGNED DEFAULT NULL
) ENGINE=InnoDB
"""

class Promotable(Role):
    def __init__(self, repl_user, master):
        self.__master = master
        self.__user = repl_user
```

```
    def _add_global_id_tables(self, Master):
        master.sql(_GLOBAL_TRANS_ID_DEF)
        master.sql(_LAST_EXEC_TRANS_DEF)
    if not master.sql("SELECT @@warning_count"):
        master.sql("INSERT INTO Last_Exec_Trans() VALUES ()")

    def _relay_events(self, server, config):
        config.set('mysqld', 'log-slave-updates')

    def imbue(self, server):
        # Fetch and update the configuration
        config = server.get_config()
        self._set_server_id(server, config)
        self._enable_binlog(server, config)
        self._relay_event(server, config)

        # Put the new configuration in place
        server.stop()
        server.put_config(config)
        server.start()
        # Add tables to Master
        self._add_global_id_tables(self.__master)

        server.repl_user = self.__master.repl_user
```


为使用 global transaction ID，需要正确地常规配置 Slave 和 Master。当提交每个事务时，仍需要更新 *Last_Exec_Trans* 表。在例 4-11 中，可以看到在 PHP 中执行的一个提交事务的例子。代码是用 PHP 写的，因为它是应用程序代码的一部分，而不是管理配置的代码的一部分。

例4-11：*启动、提交及中止事务的代码*

```
function start_trans($link) {
  mysql_query("START TRANSACTION", $link);
}

function commit_trans($link) {
  mysql_select_db("common", $link);
  mysql_query("SET SQL_LOG_BIN = 0", $link);
  mysql_query("INSERT INTO Global_Trans_ID() VALUES ()", $link);
  $trans_id = mysql_insert_id($link);
  $result = mysql_query("SELECT @@server_id as server_id", $link);
  $row = mysql_fetch_row($result);
  $server_id = $row[0];

  $delete_query = "DELETE FROM Global_Trans_ID WHERE number = %d";
  mysql_query(sprintf($delete_query, $trans_id),
  $link);
```

```
  mysql_query("SET SQL_LOG_BIN = 1", $link);
  $update_query = "UPDATE Last_Exec_Trans SET server_id = %d, trans_id = %d";
  mysql_query(sprintf($update_query, $server_id, $trans_id), $link);
  mysql_query("COMMIT", $link);
}

function rollback_trans($link) {
  mysql_query("ROLLBACK", $link);
}
```

可以用这段代码来提交事务，通过调用函数取代通常的 COMMIT 和 ROLLBACK。例如，可以写一个 PHP 函数来添加信息到数据库或更新用户计数器的信息。

```
function add_message($email, $message, $link) {
  start_trans($link);
  mysql_select_db("common", $link);
  $query = sprintf("SELECT user_id FROM user WHERE email = '%s'", $email);
  $result = mysql_query($query, $link);
  $row = mysql_fetch_row($result);
  $user_id = $row[0];

  $update_user = "UPDATE user SET messages = messages + 1 WHERE user_id =
  %d";
  mysql_query(sprintf($update_user, $user_id), $link);
  $insert_message = "INSERT INTO message VALUES (%d,'%s')";
  mysql_query(sprintf($insert_message, $user_id, $message), $link);
  commit_trans($link);
}
$conn = mysql_connect(":/var/run/mysqld/mysqld1.sock", "root");
add_message('mats@example.com', "MySQL Python Replicant rules!", $conn);
```

138

剩下的任务就是在出现故障时处理实际的 Slave 的升级。升级程序已经在上面概述过了，但它的执行更复杂。第一步是取得远程的二进制日志文件，与第 2 章中用的方法类似，需要取得整个二进制日志文件，因为并不知道从哪里开始读取。例 4-12 中的 fetch_remote_binlog 函数返回一个二进制日志的行迭代器。

例4-12：获取远程的一个二进制日志

```
def fetch_remote_binlog(server, binlog_file):
command = ["mysqlbinlog",
           "--read-from-remote-server",
           "--force",
           "--host=%s" % (server.host),
           "--user=%s" % (server.sql_user.name)]
if server.sql_user.passwd:
    command.append("--password=%s" % (server.sql_user.passwd))
command.append(binlog_file)
return iter(subprocess.Popen(command, stdout=subprocess.PIPE).stdout)
```

迭代器一个接一个地返回二进制日志的行，因此这些行必须进一步分解成事务和事件，以便更方便地处理二进制日志。例 4-13 中，你可以看到 group_by_event 函数将同属于一个事件的行组成一个字符串，group_by_trans 函数将一串事件（被 group_by_event 返回的）组成一列，每列代表一个事务。

例4-13：分析mysqlbinlog的输出并提取出事务

```
delimiter = "/*!*/;"

def group_by_event(lines):
    event_lines = []
    for line in lines:
        if line.startswith('#'):
            if line.startswith("# End of log file"):
                del event_lines[-1]
                yield ''.join(event_lines)
                return
            if line.startswith("# at"):
                yield ''.join(event_lines)
                event_lines = []
        event_lines.append(line)

139 def group_by_trans(lines):
    group = []
    in_transaction = False
    for event in group_by_event(lines):
        group.append(event)
        if event.find(delimiter + "\nBEGIN\n" + delimiter) >= 0:
            in_transaction = True
        elif not in_transaction:
            yield group
            group = []
        else:
            p = event.find("\nCOMMIT")
            if p >= 0 and (event.startswith(delimiter, p+7)
                           or event.startswith(delimiter, p+8)):
                yield group
                group = []
                in_transaction = False
```

例 4-14 展示了 scan_logfile 函数，为被引入的 global transaction ID 扫描 mysqlbinlog 的输出。这个函数接受从中获取二进制日志文件的 Master、需要扫描的二进制日志文件的名字（文件名是 Master 上的二进制日志文件名），以及 on_gid 回调函数，每当一个 global transaction ID 被看见时 on_gid 都将被调用。on_gid 函数在被调用时传入 global transaction ID（由一个 server_id 和一个 trans_id 组成）和事务末尾的二进制日志位置。

例4-14：为global transaction ID扫描二进制日志文件

```
_GIDCRE = re.compile(r"^UPDATE Last_Exec_Trans SET\s+"
                     r"server_id = (?P<server_id>\d+),\s+"
                     r"trans_id = (?P<trans_id>\d+)$", re.MULTILINE)
_HEADCRE = re.compile(r"#\d{6}\s+\d?\d:\d\d:\d\d\s+"
                     r"server id\s+(?P<sid>\d+)\s+"
                     r"end_log_pos\s+(?P<end_pos>\d+)\s+"
                     r"(?P<type>\w+)")

def scan_logfile(Master, logfile, on_gid):
    from mysqlrep import Position
    lines = fetch_remote_binlog(master, logfile)
    # Scan the output to find global transaction ID update statements
    for trans in group_by_trans(lines):
        if len(trans) < 3:
            continue
        # Check for an update of the Last_Exec_Trans table
        m = _GIDCRE.search(trans[-2])
        if m:
            server_id = int(m.group("server_id"))
            trans_id = int(m.group("trans_id"))
            # Check for an information comment with end_log_pos. We
            # assume InnoDB tables only, so we can therefore rely on
            # the transactions to end in an Xid event.
            m = _HEADCRE.search(trans[-1])
            if m and m.group("type") == "Xid":
            pos = Position(server_id, logfile, int(m.group("end_pos")))
            on_gid(server_id, trans_id, pos)
```

140

最后一步的代码已经在例 4-15 中给出了。`promote_slave` 函数取得丢失了 Master 的 Slave 列表，并升级 Slave 列表中的新 Master。最后，通过扫描二进制日志重新连接所有其他的 Slave 到升级的 Slave。使用支持函数 `fetch_global_trans_id` 的代码从我们介绍过的表中获取 global transaction ID。

例4-15：确定新的Master并将所有的Slave重新连接到新的Maste

```
def fetch_global_trans_id(slave):
    result = slave.sql("SELECT server_id, trans_id FROM Last_Exec_Trans")
    return (int(result["server_id"]), int(result["trans_id"]))

def promote_slave(slaves):
    slave_info = {}

    # Collect the global transaction ID of each slave
    for slave in slaves:
        slave.connect()
        server_id, trans_id = fetch_global_trans_id(slave)
        slave_info.setdefault(trans_id, []).append((server_id, trans_id, slave))
        slave.disconnect()
```

```
# Pick the slave to promote by taking the slave with the highest
# global transaction id.
new_master = slave_info[max(slave_info)].pop()[2]

def maybe_change_master(server_id, trans_id, position):
    from mysqlrep.utility import change_master
    try:
        for sid, tid, slave in slave_info[trans_id]:
            if slave is not new_master:
                change_master(slave, new_master, position)
    except KeyError:
        pass

# Read the the master logfiles of the new master.
new_master.connect()
logs = [row["Log_name"] for row in new_master.sql("SHOW MASTER LOGS")]
new_master.disconnect()

# Read the master logfiles one by one in reverse order, the
# latest binlog file first.
logs.reverse()
for log in logs:
    scan_logfile(new_master, log, maybe_change_master)
```

在该代码中值得一提的是，Slave 被收集到一个字典中，使用来自 global transaction ID 的 transaction ID 作为关键字。由于可能存在若干个 Slave 关联相同的关键字，我们使用 Alex Martelli et al 的“*Python cookbook*”（O'Reilly）中的“字典中的每个关键字关联
141 多个值”方法。它列举了每个关键字下的服务器列表，而且可以快速查找和处理仅基于 transaction ID 的 maybe_change_master。



例 4-15 中的代码并不保证 transaction ID 会按顺序排列，因此如果很重要，必须采取额外的方法。如果一个事务在获取 global transaction ID 时，在其可以提交另一个事务（即获取 global transaction ID 和提交的事务）之前被打断了，那么 transaction ID 可能是无序的。为确保 transaction ID 反映事务开始的顺序，只需在获取 global transaction ID 之前添加 SELECT ... FOR UPDATE，改变代码如下：

```
def commit_trans(cur):
    cur.execute("SELECT * FROM Last_Exec_Trans FOR UPDATE")
    cur.execute("SET SQL_LOG_BIN = 0")
    cur.execute("INSERT INTO Global_Trans_ID() VALUES ()")
    .
    .
    cur.commit()
```

这将锁住该行直到事务被提交，但也会使系统运行变慢，如果不需要排序，那么这就是浪费。

提交事务的存储过程

本章中同步服务器的主要方法是在应用程序中执行事务提交程序，这意味着应用程序代码需要知道表名及如何产生和操纵 global transaction ID 的复杂性。一旦理解了，这些复杂性就不像它们起初看起来那样是障碍了。大多数情况下，可以通过在应用程序代码中创建函数来相对容易地处理它们，应用程序编码人员可以调用它而不必知道其细节。

另一个方法是把事务提交逻辑通过存储过程放到数据库服务器中，视情况而定，这样有时会是一个更好的选择。

例如，提交程序可以被更改而不需要修改应用程序代码。

为使这项技术可行，必须将 *Global_Trans_ID* 表的 transaction ID 和 server ID 放到存储例程中的用户定义变量或局部变量中。取决于你选择的方法，二进制日志中的查询看起来会略有不同。

使用局部变量是不太可能干扰周围的代码的，因为用户定义的变量从存储过程中“泄漏”出去。

提交事务的程序将是：

142

```
CREATE PROCEDURE commit_trans ()
    SQL SECURITY DEFINER
BEGIN
    DECLARE trans_id, server_id INT UNSIGNED;
    SET SQL_LOG_BIN = 0;
    INSERT INTO global_trans_id() values ();
    SELECT LAST_INSERT_ID() INTO trans_id,
           @@server_id INTO server_id;
    SET SQL_LOG_BIN = 1;
    INSERT INTO last_exec_trans(server_id, trans_id)
           VALUES (server_id, trans_id);
    COMMIT;
END
```

应用程序代码提交事务因此而简单：

```
    CALL Commit_Trans();
```

现在剩下的任务是改变程序以用于扫描二进制日志中的 global transaction ID。如何调用出现在二进制日志中的这个函数？快速调用 mysqlbinlog 如下所示：

```
# at 1724
```

```
#091129 18:35:11 server id 1 end_log_pos 1899 Query thread_id=75
  exec_time=0 error_code=0
SET TIMESTAMP=1259516111/*!*/;
INSERT INTO last_exec_trans(server_id, trans_id)
  VALUES ( NAME_CONST('server_id',1), NAME_CONST('trans_id',13))
/*!*/;
# at 1899
#091129 18:35:11 server id 1 end_log_pos 1926 Xid = 1444
COMMIT/*!*/;
```

如你所见，server ID 和 transaction ID 在输出中显然都是不可见的。如何使用正则表达式去匹配该语句，请读者自己去练习。

循环复制

在读过关于双 Master 的内容之后，你可能想知道是否有可能建立多 Master（两个以上 Master）之间相互复制的系统。由于每个 Slave 只能有单个 Master，因此只可能通过以环形的方式创建复制来得到这样的系统。

尽管不推荐这样的格局，但这当然是有可能的。不推荐的原因是：在系统发生故障时，很难让其再正常工作，造成这种情况的原因将在下面的章节中描述。

出于位置的原因，使用循环复制设置三个或更多的服务器的复制是相当实用的。举一个现实生活中的例子，考虑为整个欧洲用户提供移动电话的运营商的案例。由于手机用户
143 漫游颇多，可以很方便地为客户注册手机，因此，通过在欧洲的一些战略要地放置数据中心，就有可能快速验证通话数据，也可以在本地注册新的通话。这些更改可以被复制到环中的所有服务器，最终所有服务器将有准确的计费信息。在这种情况下，循环复制是一个完美的设置：所有的用户数据都被放置到所有的站点，并且允许在所有的数据中心进行数据更新。

建立循环复制（如图 4-10 所示）是非常简单的。例 4-16 提供了一个脚本去自动建立循环复制，那么复杂在什么地方呢？在每次建立的时候，你都要问问你自己："当出问题时会发生什么？"

例4-16：建立循环复制

```
def circular_replication(server_list):
    count = len(server_list)
    for i in range(0, count):
        change_Master(server_list[(i+1) % count], server_list[i])
```

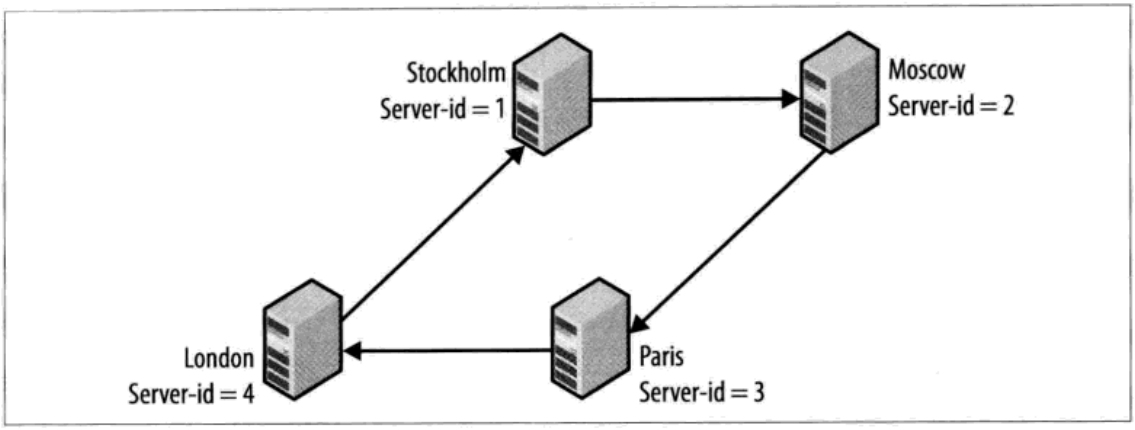


图4-10：设置循环复制

在图 4-10 中，有 4 台以所在城市命名的服务器（名称是任意挑选的，不反映真实的设置）。复制在一个环形中进行：从“Stockholm”到“Moscow”到“Paris”到“London”然后再回到“Stockholm”。这意味着“Moscow”是“Paris“的上游，但它是“Stockholm”的下游。假设“Moscow”突然意外地停止，为使复制继续，有必要重新连接“下游”的服务器“Paris”到“上游”服务器“Stockholm”，以确保系统的继续运行。

图 4-11 展示了一个场景，其中一台服务器出现故障而其他服务器重新连接以使复制继续。听起来很简单，不是吗？并不是真的像它看起来那样简单。有三个基本问题是你不得不考虑的：

- 下游的服务器（连接到已失效的 Master 的 Slave）需要连接到上游的服务器并从它最后一次看见的位置开始复制。如何决定这一位置？
- 假设崩溃的服务器在系统崩溃前已经成功发送了一些事件。这些事件将发生什么？ 144
- 需要考虑如何使故障服务器重新连入拓扑结构。如果服务器应用了它自己的一些写入二进制日志但尚未发出的事务将发生什么？很显然，这些事务都将丢失，所以需要处理这个问题。

当探测到其中一个服务器其中发生故障了时，很容易使用 CHANGE MASTER 命令来连接下游的服务器到上游服务器，但是为了复制正确地工作，必须决定正确的位置。为找到正确的位置，使用类似于在 Slave 升级中所用的二进制日志扫描技术。然而，在这种情况下，当决定从什么位置开始时，需要考虑一些服务器。之前介绍的 *Last_Exec_Trans* 表包含了从那台服务器所看到的 server ID 和 global transaction ID。

第二个问题更复杂。如果失败的服务器已经发出一个事件，就没有任何办法来从复制流中删除那个事件，因此它将永远围着复制拓扑结构绕圈子。如果语句是等幂的，它可以被重新应用多次，而不会造成问题。这种情况在短期内是可以做到的，但一般来说，该

声明需要以某种方式被删除。

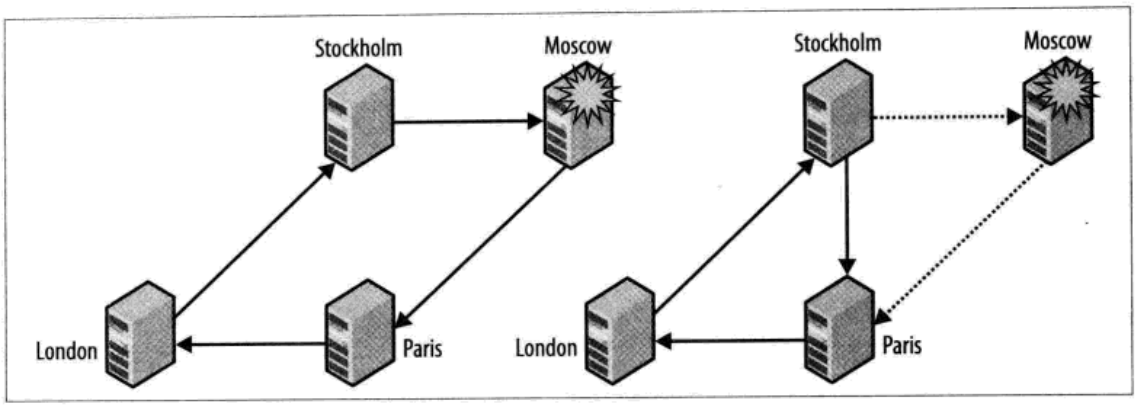


图4-11：改变拓扑结构响应失败的服务器

在 MySQL 5.5 版本中，参数 IGNORE_SERVER_IDS 被加入 CHANGE MASTER 命令。该参数
145 允许服务器从复制流中删除更多的事件，不仅仅允许删除具有与服务器相同的 Server ID 的事件。因此，如果服务器拥有图 4-11 所示的 ID，可以使用下面的命令重新连接 Paris 到 Stockholm：

```
paris > CHANGE MASTER TO
    ->   MASTER_HOST='stockholm.example.com',
    ->   IGNORE_SERVER_IDS = (2);
```

MySQL 5.5 之前的版本没有这样的支持，可能必须用其他方法来删除引起问题的事件。最简单的方法可能是让一个服务器临时具有跟崩溃的服务器相同的 ID，这样就可以删除违规事件。

假设你使用的是 MySQL 5.5，那么移除环状拓扑结构中的服务器的步骤如下：

1. 为所有仍然运行的服务器确定在下游服务器上最后提交的事务的 global transaction ID。

```
paris> SELECT Server_ID, Trans_ID FROM Last_Exec_Trans WHERE Server_ID != 2;
+-----------+----------+
| Server_ID | Trans_ID |
+-----------+----------+
| 1         |   5768   |
| 3         |   4563   |
| 4         |   768    |
+-----------+----------+
3 rows in set (0.00 sec)
```

2. 为在 *Last_Exec_Trans* 中看到的最后的 global transaction ID 扫描下游服务器的二进

制日志。

3. 使用 CHANGE MASTER 连接下游服务器到这个位置。

```
paris > CHANGE MASTER TO
     ->   MASTER_HOST='stockholm.example.com',
     ->   IGNORE_SERVER_IDS = (2);
```

由于与其他服务器相比，故障的服务器可以被替代，再次将它连接到环中最安全的方法是：从环中的某个服务器中恢复该服务器，然后重新将它连接到环中，因此新的服务器又一次在环中了。完成的步骤是：

4. 从环中一个已经存在的服务器那里恢复该服务器——该服务器将最终成为上游服务器，并将它作为一个 Slave 附着在服务器上.

```
moscow> CHANGE MASTER TO MASTER_HOST='stockholm.example.com';
Query OK, 0 rows affected (0.18 sec)
moscow> START SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

5. 一旦服务器已经充分赶上了，通过断开下游服务器的连接来打破这个环。这个服务器将不会再接收任何的更新。

```
paris> STOP SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

6. 由于恢复的服务器可能不会拥有下游服务器所拥有的一切事件，有必要等到恢复的服务器至少拥有下游服务器拥有的所有事件。由于这些位置是针对于同一个服务器而言的，你可以结合使用 SHOW SLAVE STATUS 和 MASTER_POS_WAIT 事故。

```
paris> SHOW SLAVE STATUS;
...
Relay_Master_Log_File: stockholm-bin.000096
...
Exec_Master_Log_Pos: 756648
1 row in set (0.00 sec)
moscow> SELECT MASTER_POS_WAIT('stockholm-bin.000096', 756648);
+-----------------------------------------------+
| MASTER_POS_WAIT('stockholm-bin.000096', 756648) |
+-----------------------------------------------+
|                                        985761 |
+-----------------------------------------------+
1 row in set (156.32 sec)
```

146

7. 通过扫描恢复的服务器的二进制日志获取下游服务器所看到的最后的 global ID，以确定恢复的服务器的事件位置。

8. 连接下游服务器到恢复的服务器，然后开始复制。

```
paris> CHANGE MASTER TO
    ->    MASTER_HOST='moscow.example.com',
    ->    MASTER_LOG_FILE='moscow-bin.000107',
    ->    MASTER_LOG_POS=196758,
Query OK, 0 rows affected (0.18 sec)
moscow> START SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

小结

高可用性是现实中不容忽视的概念。本章介绍了高用性，以及如何在 MySQL 中实现它。在下一章中，我们将更深入地研究高可用性，同样将审查其姊妹话题：向外扩展。

> Joel 的邮件通知铃声响起，他点击了邮件然后打开最后的信息。这是来自 Summerson 先生的，Summerson 先生对他的报告做了注释。他通读完毕然后在末尾发现了他所期望看到的："我喜欢这个想法，特别是冗余热备份的策略。让计划付诸实施。"
>
> Joel 叹了口气，他意识到熟悉同事的计划必须要缓一缓。他有太多工作要做。

MySQL集群的横向扩展

Joel站起来伸展了一下身体，心想现在是休息时间了。他绕过桌子，正要去休息室，正好在办公室门口遇到老板。“上午好，先生。”

“嗨，Joel。我们刚刚卖出了一笔新产品的许可证。市场部的人告诉我数据库负载将会至少增加10倍。”

Joel扬了扬眉毛。上周他刚加了一台Slave机器改善了负载，可是没有根本解决问题。

“我们需要横向扩展，Joel。”

“是的，先生，我马上去做。”

Summerson先生笑着拍了拍Joel的肩膀，然后顺着大厅走向他的办公室。

Joel在那站了一会儿，考虑着“横向扩展”是什么意思，该如何规划。“我得再看点资料。”他一边往休息室走一边嘀咕着。

当负载开始增加的时候（如果部署完好，那就只剩下何时开始增加的问题了），有两种方法来处理这个问题。第一种方法是购买更强大的服务器来应对增加的负载，称为纵向扩展（*scale up*），第二种方法是添加更多的服务器，称为横向扩展（*scale out*）。其中横向扩展方案更加常用，因为它通常只需购买一批低成本的标准服务器，更具有成本效益。

除了负载增加以外，添加服务器还可以支持高可用性和其他商业需求。如果有效使用，横向扩展可以将所有的服务器资源结合在一起，比如计算能力，从而极大程度地利用这些资源。

本章不打算深入硬件、网络及其他横向扩展相关的问题，这些超出了本章的范围，你可
148 以参考 Baron Schwartz 等人的“High Performance MySQL”（O'Reilly, *http://oreilly.com/catalog/9780596101718/*）一书[译注]。我们要讨论的是如何建立 MySQL 复制，才能充分利用横向扩展。首先介绍复制的基础知识，然后着手开发大量服务器上简化复制管理的 Python 库，接着考察复制如何适应组织的业务需求。

横向扩展和复制的常见用途有：

- 读操作的负载均衡
 由于 Master 忙于更新数据，所以将响应查询的服务器分离开来是明智的，因为查询只需要读取数据。使用复制将 Master 上的改变发给多个 Slave（数量根据需要而定），每个 Slave 存储当前数据并处理查询。
- 写操作的负载均衡
 高流量的部署将处理分布到很多计算机上，有时候上千。因此，复制在分布待处理信息的过程中起着关键作用。根据数据的业务用途及其使用性质不同，信息可以有很多种分布方式：
 - 基于信息的角色分布。很少更新的表放在一个服务器上，而频繁更新的表则分割到多个服务器上。
 - 按地理位置分割，这样流量可以直接定向到最近的服务器。
- 通过热备份进行灾难避免
 如果 Master 出现故障，一切都会停止，就无法执行事务（可能是关键事务）、获取客户信息或检索其他关键数据。这会严重扰乱你的业务，所以要（几乎）不惜任何代价避免这种情况发生。最简单的方法就是配置一个专门的 Slave 作为热备份，如果 Master 发生故障，随时接手 Master 的工作。
- 通过远程复制进行灾难避免
 由于灾难的存在，比如断电、地震或洪水等，每个部署都有数据中心发生故障的风险。要想缓和这种情况，地理上的远程站点之间使用复制来传输信息。
- 备份
 通常用一个额外的服务器来做备份。这样你可以在不影响 Master 的前提下执行备份，因为备份服务器是离线的，你可以做任何事情。
- 生成报表
 服务器上的数据创建报表会降低服务器的性能，有时候影响很大。如果产生报表
149 需要大量后台作业，那么创建一个专门的 Slave 来完成这项工作是很值得的。停止

译注 该书的中文版《高性能 MySQL》（978-7-121-10245-5）于 2010 年 1 月由电子工业出版社出版。

Slave 上的复制，获得数据库某个时间点上的快照，然后在其上运行大量查询，这样不会干扰主业务服务器。例如，如果在某天的最后一个事务完成后停止复制，你就可以在其他业务正常的情况下获取日报表。

- 过滤或分区数据
 如果网络连接很慢，或者部分数据对于某些客户端不可用，可以添加一个服务器进行数据过滤处理。当数据需要分区到独立的服务器上时，这种方法同样有用。

读操作的横向扩展

一定要理解下面这种方法（添加 Slave）仅仅扩展读，而不能做到写操作的横向扩展。每个新加的 Slave 都要处理与 Master 相同的写负载。整个系统的平均负载可以描述为：

$$AuerageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity}$$

所以，如果单个服务器每秒有 10 000 的事务量，而 Master 每秒的写负载为 4 000 个事务，那么每秒的读负载为 6 000，结果就是：

$$AuerageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity} = \frac{6000 + 4000}{10000} = 100\%$$

现在，如果添加 3 个 Slave，每秒的事务量增加到 40 000。因为写操作也会被复制，每个写操作都会执行 4 次（一次在 Master 上，其他 3 个 Slave 各执行一次），这样每个 Slave 的写负载就是每秒 4 000 个事务。因为在 Slave 之间分布开来了，所以总的读负载并没有增加。现在的平均负载就是：

$$AuerageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity} = \frac{6000 + 4 \times 4000}{4 \times 10000} = 55\%$$

注意公式中总事务量增加了 4 倍，因为我们一共有 4 台服务器，而且复制使写负载同样也增加了 4 倍。

经常容易忘记的是，所有 Master 的写查询都会复制到每个 Slave。所以用这种简单的方法并不能扩展写操作，而只能实现读操作的横向扩展。本章后面将会使用一种称为 *sharding* 的技术实现写操作的横向扩展。

150

异步复制的价值

MySQL 复制是异步的，这种类型的复制特别适合现代应用，如网站等。

为了处理大量的读请求，站点通过复制创建 Master 的拷贝，然后让 Slave 处理所有的读请求，而 Master 处理写请求。这个复制过程是异步的，因为 Master 并不等待 Slave 应用改变，而是将每个更新请求分发到 Slave，并假定它们最终会达到一致且复制所有的改变。这种提高性能的技术通常也是横向扩展的好办法。

相反，同步复制会保持 Master 和 Slave 之间的同步，如果 Slave 不同意事务提交，则 Master 也不能提交这个事务。也就是说，同步复制要求 Master 必须等待所有 Slave 的写操作完成。

异步复制比同步复制快得多，下面我们将详细描述原因。与异步复制相比，同步复制需要额外的同步机制来保证一致性，一般通过两阶段提交协议来实现。两阶段提交协议保证了 Master 和 Slave 之间的一致性，但却需要它们之间有额外的通信消息传递，其一般工作过程如下：

1. 当执行提交语句时，事务被发送到 Slave，Slave 开始准备事务的提交。
2. 每个 Slave 都要准备事务，然后向 Master 发送 OK（或 ABORT）消息，表明事务已经准备好（或者无法准备该事务）。
3. Master 等待所有 Slave 发送 OK 或 ABORT 消息。
 a. 如果 Master 收到所有 Slave 的 OK 消息，它就会向所有 Slave 发送提交消息，告诉 Slave 提交该事务；
 b. 如果 Master 收到来自任何一个 Slave 的 ABORT 消息，它就向所有 Slave 发送 ABORT 消息，告诉 Slave 去中止事务。
4. 每个 Slave 等待来自 Master 的 OK 或 ABORT 消息。
 a. 如果 Slave 收到提交请求，它们就会提交事务，并向 Master 发送事务已提交的确认；
 b. 如果 Slave 收到取消请求，它们就会撤销所有改变并释放所占有的资源，从而中止事务，然后向 Master 发送事务已中止的确认。
5. 当 Master 收到来自所有 Slave 的确认后，就会报告该事务被提交（或中止），然后继续进行下一个事务处理。

151 这个协议之所以慢，原因是它一共需要 4 次消息传递，包括事务消息和准备请求的消息。

主要问题不在于处理同步所需的网络通信量，而是网络延迟和 Slave 处理提交的延迟，而且在所有 Slave 确认事务之前 Master 的提交会被阻塞。而异步复制只需要一个事务消息即可，这样的好处是，Master 不需要等待 Slave 的处理就可以立即报告事务的提交，从而极大地提高了性能。

那么，同步复制中，Slave 的处理为什么会阻塞事务的提交呢？如果 Slave 离 Master 很近，同步复制所需的额外消息传递不会带来什么影响，但是如果 Slave 很远——可能在另一个城镇甚至另一个洲——结果就大不相同了。

表 5-1 显示了服务器每秒处理 10 000 个事务提交的示例。提交时间为 0.1ms（请注意有些实现中，比如 MySQL 集群，在事务间独立时，可以并行处理多个事务的提交）。如果网络延迟为 0.01ms（通过 ping 我们自己的电脑，得到这个值作为基线），事务提交时间就会增加为 0.14ms，即每秒约 7 000 个事务。如果网络延迟为 10ms（通过 ping 附近城市的服务器得到这个值），事务提交时间就会增加到 40.1ms，即每秒约 25 个事务！相反，异步复制则没有任何延迟，因为事务被立即提交，所以事务提交时间保持不变，仍为最初的每秒 10 000 个事务，就好像没有 Slave 一样。

表5-1：同步复制引起性能降低的典型例子

Latency(ms)	Transaction commit time(ms)	Equivalent transactions persecond	Examplecase
0.01	0.14	~ 7,100	Same computer
0.1	0.5	~ 2,000	Small LAN
1	4.1	~ 240	Bigger LAN
10	40.1	~ 25	Metropolitan network
100	400.1	~ 2	Satellite

异步复制的性能提升是以牺牲一致性为代价的。回想一下，异步复制中事务的提交是立即报告的，而不等待任何来自 Slave 的确认。这意味着 Slave 尚没有提交的事务，Master 可能也会认为该事务已提交。事实上，事务甚至没有离开 Master，却仍在等待被发送到 Slave。

有两个问题需要注意：

- 如果 Master 出现故障，事务就会“消失”；
- Slave 上执行的查询可能会返回旧数据。

后面我们会谈到如何保证读取的是当前数据，目前只要记住异步复制中有些问题需要我们去处理即可。

管理复制拓扑

通过创建新 Slave，然后将它们添加到拓扑中来进行扩展部署。术语“复制拓扑”是指使用复制的方式连接服务器。图 5-1 给出了复制拓扑的示例：简单拓扑、树形拓扑、双主拓扑和环形拓扑。

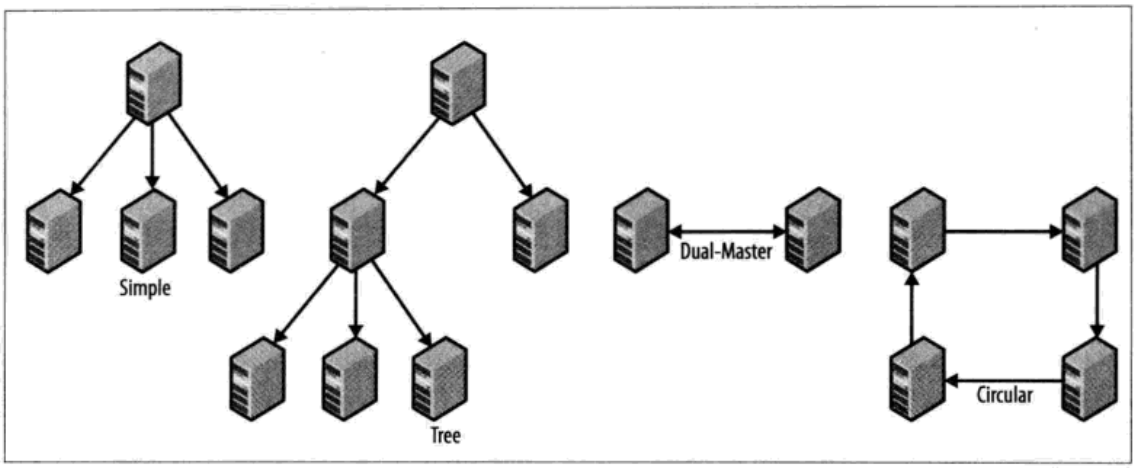


图5-1：简单拓扑、树形拓扑、双主拓扑和环形拓扑

这些拓扑用途各不相同：例如，双主拓扑很好地处理故障转移；环形复制和双主拓扑允许不同站点各自在本地工作，然后将改变复制到其他站点。

简单和树形拓扑用于横向扩展。使用复制导致读操作的数量远远超过了写，使得部署上有两种特殊要求：

- 需要负载均衡
 这里使用术语“负载均衡”描述服务器之间查询划分的方式。复制说明了为负载均
153 衡原因也提供了解决方法。首先，复制将写操作定向到 Master，读操作交给 Slave，这样可以做到基本划分负载。而且，有时候必须向特定 Slave 发送特定的查询。
- 需要管理拓扑
 服务器总会有崩溃的时候，这样就必须要更换它们。Slave 崩溃可能并不急着换，但是必须尽快更换崩溃的 Master。
 另外，如果 Master 崩溃了，客户端将会被重定向到新的 Master。如果 Slave 崩溃，必须将这个 Slave 移出负载均衡池，保证不再有查询请求发给它。

为处理负载均衡和管理，要使用一些工具来管理复制拓扑，特别是监控服务器状态和性能的工具和处理分布式查询的工具。

为了使负载均衡更加高效，服务器要保留空闲处理能力，原因是：

- 峰值负载处理
 要有余地处理峰值负载。系统负载从来不是均匀变化的，而是上下波动的。这种空闲处理能力取决于应用程序，所以需要在响应时间异常时密切监控应用。
- 分布成本
 运行复制需要空闲处理能力。在分布式系统中复制总会产生某些“浪费”，包括管理分布式系统所需的额外查询，例如需要额外的查询来确定哪个节点执行这个读查询。容易忽略的一点是，每个 Slave 都要执行和 Master 相同的写操作。来自 Master 的查询按照一定的顺序执行，不会产生更新冲突，但为此 Slave 需要付出额外的处理能力来完成复制。
- 管理性任务
 重新构建复制需要空闲处理能力，因此需要临时性干点其他的活儿，例如，在服务器之间移动数据的时候。

负载均衡有两种基本的工作方式：一种是基于查询类型请求服务器的应用，另一种是中间层（通常指代理）分析查询，然后发给适当的服务器。

使用中间层分析和分布查询（如图 5-2 所示）是目前最灵活的方法，但有两个不足：

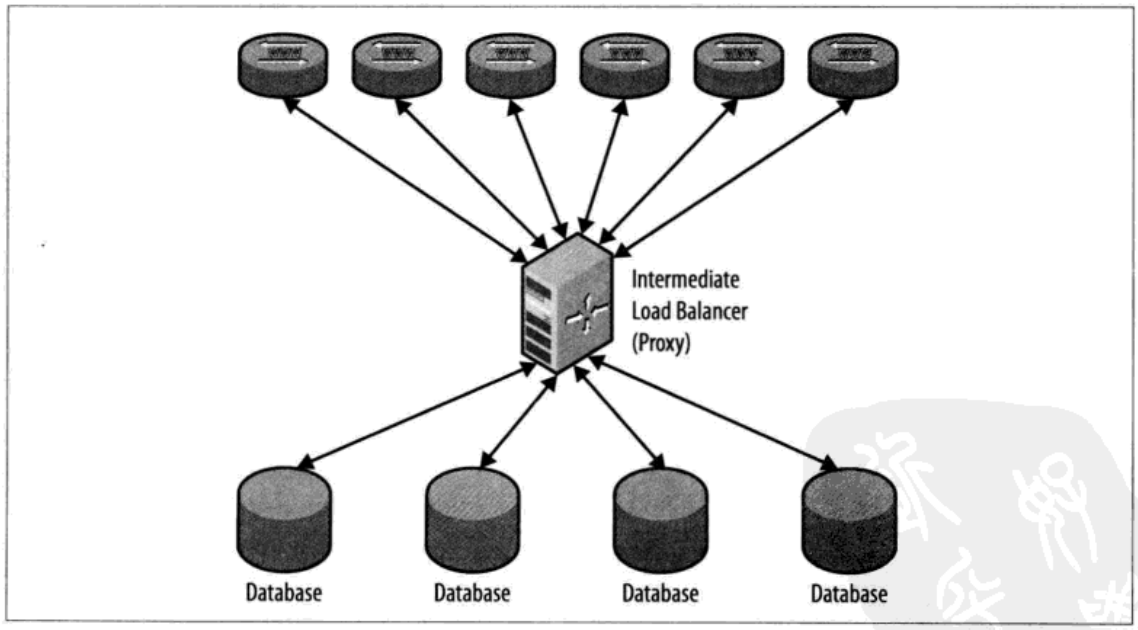


图5-2：使用代理分发查询

- 分析查询需要消耗处理资源。这会延迟查询，因为查询会被解析和分析两次：一次

154 是代理，另一次是 MySQL 服务器。分析越高级，查询延迟就越大。根据应用的不同，这种延迟也许是个问题，也许不是。

- 正确的查询分析很难实现，有时甚至不可能实现。代理通常向应用程序员隐藏了部署的内部结构，以便他不用考虑部署选择。但是正因为这样，很难正确地分析客户端所发出的查询，而且可能需要在发给服务器之前进行比较大的重写。

用于代理负载均衡的工具之一就是 MySQL Proxy，它包含 MySQL 客户端协议的完整实现，所以它既可以作为服务器用于客户端连接，也可以作为客户端连接到 MySQL 服务器，也就是说是完全透明的，客户端不用区分代理和真正的服务器。

MySQL Proxy 由 Lua 语言控制。它内建了 Lua 引擎，执行小（有时候也并不那么小）程序，来拦截和操纵查询及其结果集。由于使用真实的编程语言控制，所以代理可以执行各种复杂的任务，包括查询分析、查询过滤、查询操作和查询分布。

MySQL Proxy 的配置和编程超出了本书的范围，但网上有很多这方面的出版物。其中我们认为比较有用的如下：

155 *http://dev.mysql.com/tech-resources/articles/proxy-gettingstarted.html*

Giuseppe Maxia 介绍 MySQL Proxy 的经典文章“Getting Started with MySQL Proxy”；

http://forge.mysql.com/wiki/MySQL_Proxy

MySQL Forge wiki 页面有很多关于代理的信息，包括很多参考文献和示例；

http://forge.mysql.com/wiki/MySQL_Proxy_RW_Splitting

描述 MySQL Forge 如何使用 MySQL Proxy 进行读 / 写分离，即将读查询发给一组服务器，写查询发给 Master。

使用代理的具体方法完全取决于代理类型，所以这里不讨论这个问题，而是集中研究负载均衡器在应用层的使用。负载均衡器有多种类型，包括：

- 硬件；
- 简单软件负载均衡器，如 Balance（*http://www.inlab.de/balance.html*）；
- 基于对等系统（Peer-based systems），如 Wackamole（*http://www.backhand.org/wackamole/*）；
- 完全成熟的集群方案，如 Linux Virtual Server（*http://www.linuxvirtualserver.org/*）。

还可以在 DNS 级别分散负载，或者直接在应用程序中处理负载的分散。

应用层负载均衡器示例

下面设计并实现一个简单的应用层负载均衡器，看看它是如何工作的。本节中将实现读写分离。后面会扩展这个负载均衡器以处理数据分割。

应用层负载均衡最直接的办法就是让应用程序根据要发送的查询类型向负载均衡器发起一个连接。大部分情况下，应用程序事先就知道查询是读查询还是写查询，以及会用到哪些表。其实，在设计查询时让应用程序开发者考虑这些问题可以带来其他好处，通常可以提高系统的整体性能。这样，负载均衡器连接到正确的服务器，然后应用程序就可以执行查询了。

应用层的负载均衡器需要有一个中心数据存储，存储服务器及其能够处理的查询信息。应用层的函数将查询发送到这个中心存储，返回要查询的 MySQL 服务器的名字和 IP 地址。

下面开发一个如图 5-3 所示的简易应用层负载均衡器。表示层逻辑使用 PHP，因为 PHP 广泛用于 Web 服务器。需要编写更新服务器池信息和从池中获取服务器的方法。 156

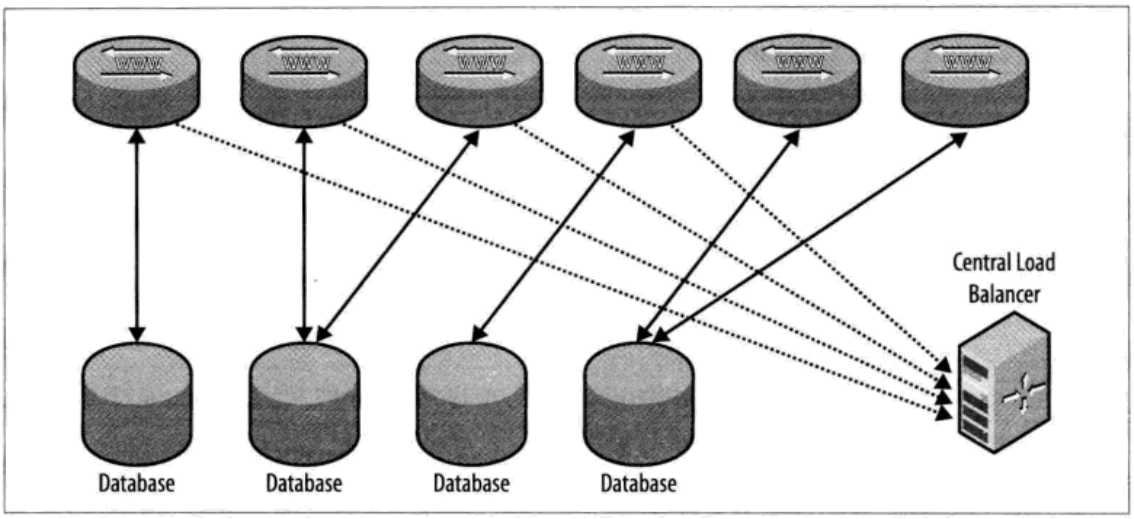


图5-3：应用层的负载均衡

这个池是这样实现的：在数据库上创建一个表存储所有服务器的相关信息，并在所有节点之间共享这个表。这样，就可以使用主机和端口作为该表的主键（而不是创建 hostID 字段），然后创建一个含有共享数据表的数据库。



中心存储需要复制，这样才不会产生单点故障。而且，因为可用服务器列表通常不会变化，所以负载均衡信息很适合缓存。

为了简单起见，避免引入对其他系统的依赖性，我们使用纯 MySQL 实现来展示应

用层负载均衡器。

你也可以使用其他非 MySQL 技术，最常用的是使用轮换调度的 DNS，或者使用分布式内存键值存储 Memcahed。

还要注意应该避免增加额外查询可能给高性能系统带来的显著开销。

157 负载均衡器将服务器目录放在负载均衡器的池中，根据它们所处理的查询类型的分成不同类别。池中的服务器信息存储在一个中心存储中。具体实现包括：数据库表（例 5-1），应用程序查询负载均衡的 PHP 方法（例 5-2），以及更新服务器信息的 Python 方法（例 5-3）。

例5-1：负载均衡器的数据库表

```
CREATE TABLE nodes (
  host CHAR(28) NOT NULL,
  port INT UNSIGNED NOT NULL,
  sock CHAR(64) NOT NULL,
  type SET('READ','WRITE') NOT NULL DEFAULT '',
  PRIMARY KEY (host, port)
);
```

存储每个主机是否可读、可写的相关信息，将它们存在 type 字段中。如果将该字段置为空，可以将服务器离线，这对于维护工作很重要。

简单的 SELECT 语句可以查找所有接受查询的服务器。由于我们只需要一个服务器，在 SELECT 语句后面加上 LIMIT 修饰符，将输出限制为一行，然后使用 ORDER BY RAND() 修饰符，将查询平均地分散到所有可用服务器上。



使用 ORDER BY RAND() 修饰符需要服务器将表中的行排序，所以取随机数可能不是最有效的方法（实际上，取随机数是个糟糕的办法）。但是这里我们仍然用这种方法，仅仅作为示例展示。

例 5-2 显示了连接服务器的 PHP 函数 getServerConnection，返回适合某个查询的服务器连接，或者如果没有找到合适的服务器，就返回 NULL。帮助函数 connect_to 负责构建连接字符串，包括主机名、端口号和 UNIX 套接字。如果是本地主机（local host），我们就用套接字连接服务器，这样更高效。

例5-2：查询负载均衡器的PHP函数

```
function connect_to($host, $port, $socket) {
  $db_server = $host == "localhost" ? ":{$socket}" : "{$host}:{$port}";
  return mysql_connect($db_server, 'query_user');
}
$COMMON = connect_to(host, port, socket);
```

```
mysql_select_db('common', $COMMON);

define('DB_WRITE', 'WRITE');
define('DB_READ', 'READ');

function getServerConnection($queryType)
{
  global $COMMON;
  $query = <<<END_OF_SQL
SELECT host, port, sock FROM nodes
 WHERE FIND_IN_SET('$queryType', type)
ORDER BY RAND() LIMIT 1
END_OF_SQL;
  $result = mysql_query($query, $COMMON);
  if ($row = mysql_fetch_row($result))
    return connect_to($row[0], $row[1], $row[2]);
  return NULL;
}
```

158

最后需要提供添加和删除服务器及实现服务器更新的实用函数。由于这些任务主要用于管理逻辑，因此利用 Python 语言的 Replicant 库来实现这个功能。这个实用程序由以下三个函数组成：

- `pool_add(common, server, type)`
 向池中添加服务器。池存储在标记为 `common` 服务器上，`type` 使用一组值列表（或其他可迭代的类型）来设置。
- `pool_del(common, server)`
 从池中删除一个服务器。
- `pool_set(common, server, type)`
 改变服务器的类型。

例5-3：负载均衡器的管理性函数

```
class AlreadyInPoolError(replicant.Error):
    pass
_INSERT_SERVER = """
INSERT INTO nodes(host, port, sock, type)
    VALUES (%s, %s, %s, %s)"""
_DELETE_SERVER = "DELETE FROM nodes WHERE host = %s AND port = %s"

_UPDATE_SERVER = "UPDATE nodes SET type = %s WHERE host = %s AND port = %s"

def pool_add(common, server, type=[]):
    common.use("common")
    try:
        common.sql(_INSERT_SERVER,
                   (server.host, server.port, server.socket, ','.join(type)));
```

```
        except MySQLdb.IntegrityError:
            raise AlreadyInPoolError

def pool_del(common, server):
    common.use("common")
159 >   common.sql(_DELETE_SERVER, (server.host, server.port))

def pool_set(common, server, type):
    common.use("common")
    common.sql(_UPDATE_SERVER, (','.join(type), server.host, server.port))
```

这些函数的使用方法如以下示例所示：

```
pool_add(common, master, ['READ', 'WRITE'])

for slave in slaves:
    pool_add(common, slave, ['READ'])
```

级联复制（Hierarchal Replication）

尽管 Master 可以很好地处理大量的 Slave，但在负载均衡器超出负荷之前（用户说最多需要 70 个 Slave，但要认识到，实际应用可能需要更多），Master 能处理的 Slave 数量却是有限的，而且常常出现 Master 无响应的问题。这种情况下，需要额外添加 Slave（一个或多个）作为中继 Slave（或简称中继服务器，*relay*），其目的是通过管理一群 Slave 来减轻 Master 上的复制负载。这种使用中继的方式称为级联复制。图 5-4 描述了一个典型设置，包括一个 Master，一个中继服务器和几个连接到中继服务器的 Slave。

默认情况下，来自 Master 的改变不会写入 Slave 的二进制日志中，所以如果按照之前的
160 > 设置在 Slave 上执行 `SHOW BINLOG EVENTS` 命令，binlog 中并没有任何事件。原因是为了记录变化而浪费磁盘空间是没有意义的：如果出现问题，比如说 Slave 崩溃，可以通过克隆 Master 或其他 Slave 来恢复它。

另一方面，中继服务器需要二进制日志保存所有改变，因为中继服务器需要把改变传给其他 Slave。然而，与普通 Slave 不同的是，中继服务器本身并不需要应用这些改变，因为它不响应任何查询。

简而言之，普通 Slave 需要将改变应用到数据库中，但不需要二进制日志。中继服务器需要保存二进制日志，但并不应用改变。

必须用表才能执行语句，但为了避免数据改变被写入数据库，应该丢弃它们。为此，建立一个称为 Blackhole 的存储引擎，接受所有语句并保证这些语句总是成功地执行，但任何数据改变都将被丢弃。中继服务器引入了额外延迟，与直接连接 Master 相比，这使

得 Slave 更加落后于 Master。应该权衡这种滞后与 Master 减少的负载量，因为级联设置的管理比简单设置困难得多。

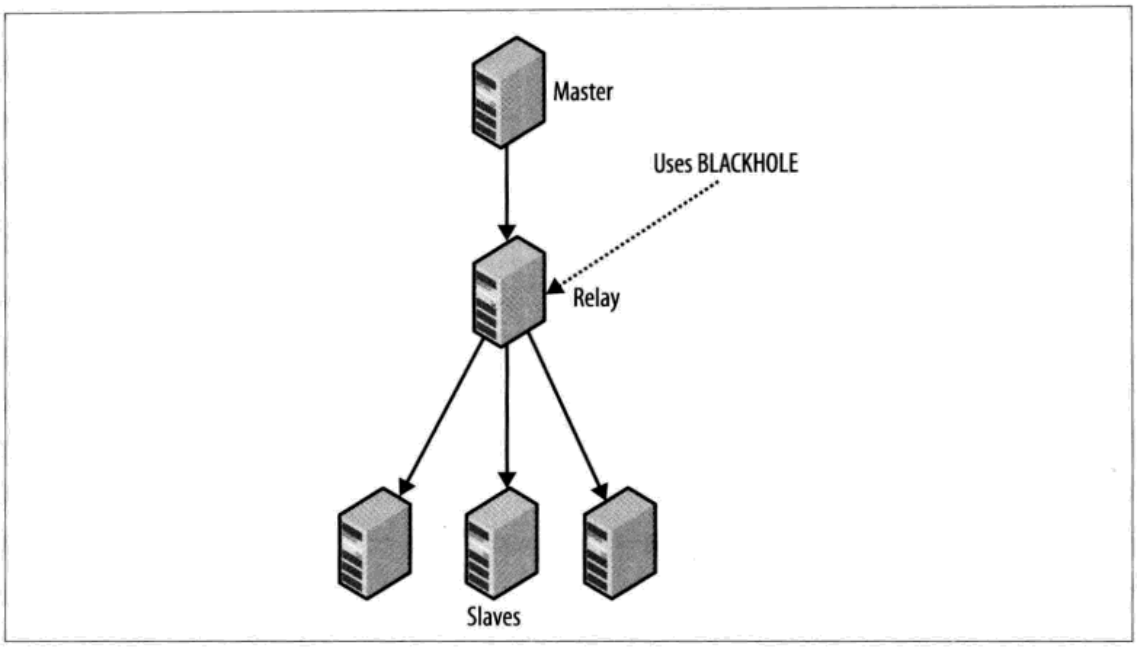


图5-4：包含Master，中继，Slave的级联拓扑

中继服务器的设置

设置中继 Slave 相当容易，但我们需要考虑不同类型的表上可以执行哪些操作，包括在中继服务器上新创建的表，以及服务器角色转变为中继服务器之前就已经存在的表。数据库中不保存数据可以加快事件的处理，并减少复制过程中 Slave 的滞后，因为没有数据更新。要设置一个中继 Slave，需要：

1. 配置 Slave，通过将 Slave 线程事件写入中继 Slave 的 binlog 中，实现 Slave 事件的发送；
2. 改变中继 Slave 上所有表的存储引擎，使用 BLACKHOLE 存储引擎保留空间和提高性能；
3. 保证中继服务器上任何新加的表都使用 BLACKHOLE 引擎。

前面提过，通过向 *my.cnf* 文件中添加 `log-Slave-updates` 选项，配置中继服务器发送 Slave 线程事件。

为了保证所有在中继 Slave 上创建的表都使用 BLACKHOLE 引擎，连接到服务器，按下面的方法设置默认存储引擎：

```
relay> SET GLOBAL STORAGE_ENGINE = 'BLACKHOLE';
```

161 最后就是使用ALTER TABLE语句将中继Slave上所有已经存在的表的存储引擎改成BLACKHOLE。由于ALTER TABLE语句不应该被写入二进制日志（至少我们希望Slave忽略它们接收到的所有改变），所以执行该语句时要临时关闭二进制日志。见例5-4。

例5-4：改变数据表的存储引擎

```
relay> SHOW TABLES FROM windy;
+-----------------+
| Tables_in_windy |
+-----------------+
| user_data       |
      .
      .
      .
| profile         |
+-----------------+
45 row in set (0.15 sec)
relay> SET SQL_LOG_BIN = 0;
relay> ALTER TABLE user_data ENGINE = 'BLACKHOLE';
         .
         .
         .
relay> ALTER TABLE profile ENGINE = 'BLACKHOLE';
relay> SET SQL_BIN_LOG = 1;
```

你需要做的就是将服务器转变为中继服务器。通常一开始所有Slave都直接连接到Master，一段时间之后发现需要引入中继Slave。原因通常是Master负载过重，但也有一些架构因素促使这种变化。那么，怎么处理这个转变呢？

可以使用前面章节所学的东西，修改已有的部署，引入新的中继服务器：

1. 将中继Slave连接到Master，并将其角色配置为中继服务器；

2. 逐个将Slave切换到中继服务器上。

使用Python添加中继服务器

现在扩展程序库以进行中继服务器管理任务的开发。服务器上已经创建了很多角色，现在我们为中继服务器定义一个特殊的角色来使用它们，如例5-5所示。

例5-5：为中继服务器定义角色

```
class Relay(role.Base):
    def __init__(self, Master):
        pass
```

```
def imbue(self, server):
    config = server.get_config()
    self._set_server_id(server, config)
    self._enable_binlog(server, config)
    config.set('mysqld', 'log-Slave-updates' '1')
    server.put_config(config)
    server.sql("SET SQL_LOG_BIN = 0")
    for db in list of databases:
        for table in server.sql("SHOW TABLES FROM %s", (db)):
            server.sql("ALTER TABLE %s.%s ENGINE=BLACKHOLE", (db,table))
    server.sql("SET SQL_LOG_BIN = 1")
```

162

专用Slave

简单的横向扩展部署（就像目前描述的那样）中，所有 Slave 都接收所有数据，从而处理任何类型的查询。但是，数据的访问频度是不一致的，通常有些数据需要频繁访问，而有些则很少被访问。例如，电子商务站点：

- 产品目录几乎总被浏览；
- 可能不会经常访问产品库存数据；
- 用户数据不太经常被访问，因为大部分重要信息是以浏览器 cookie 的形式存储的特定会话信息；
- 另一方面，如果禁用 cookie，那么几乎每个页面请求都会从服务器请求会话数据；
- 新添加的项目比旧项目更常被访问，例如，“特别优惠”可能比其他项访问得更频繁。

显然，每个 Slave 上都存储那些很少访问的数据是一种资源浪费，按照图 5-5 那样部署更好，即若干服务器专门存储很少访问的数据，另外一组服务器专门存储频繁访问的数据。

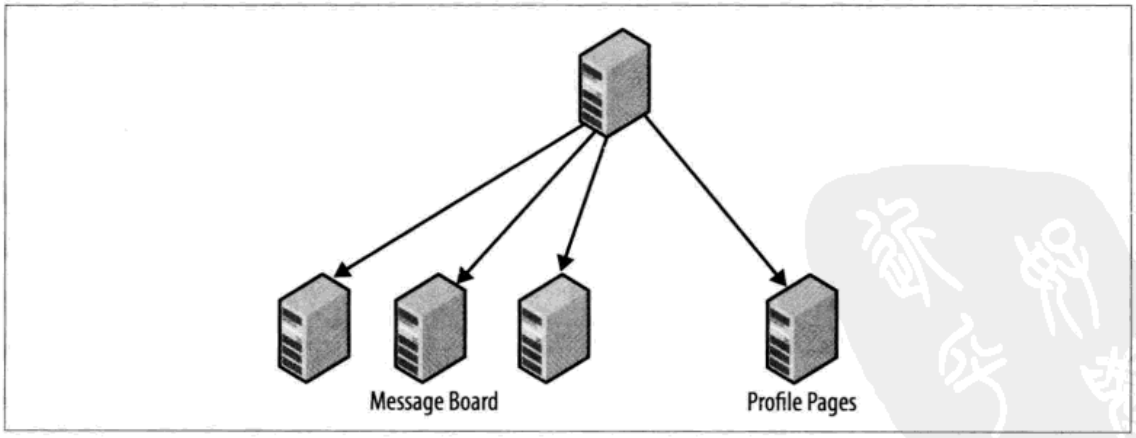


图5-5：Master和专用Slave的复制拓扑

为此，需要在复制的时候分离表。MySQL 通过过滤离开 Master 或发给 Slave 的事件实现。

163

过滤复制事件

有两种过滤事件的方法：在 Master 上过滤事件的 Master 过滤器，以及在 Slave 上过滤事件的 Slave 过滤器。Master 过滤器控制二进制日志的写入，以及发送给 Slave 的事件，而 Slave 过滤器控制 Slave 上执行的事件。对 Master 过滤器来说，已经被过滤的表事件不会写入二进制日志，Master 将事件写入二进制日志并发送给 Slave，直到事件执行时 Slave 过滤器才进行过滤。

如果使用 Master 过滤器，事件就不会存储在二进制日志中。

这意味着不可能使用 PITR 来正确地恢复数据库——如果数据库以备份映像的形式存储，恢复备份时它们会还原，但数据库表的改变无法恢复，因为二进制日志中没有记录这些改变。

如果使用 Slave 过滤器，所有改变都会通过网络传送。

显然这样会浪费带宽，特别是持久的网络连接。

本章后面将详细讨论 Master 和 Slave 过滤器的优点，以及保证二进制日志完整又能节省网络带宽的方法。

Master过滤器

创建 Master 过滤器需要配置两个选项：

```
binlog-do-db=db
```

如果当前数据库是 db，则会将该语句写入二进制日志，否则忽略。

```
binlog-ignore-db=db
```

如果当前数据库是 *db*，就忽略该语句，否则写入二进制日志。

如果想复制除了数据库的所有东西，使用 `binlog-ignore-db` 选项；如果仅仅复制数据库，就使用 `binlog-do-db` 选项。不推荐同时使用两种选项，因为逻辑上很难判断是否要复制数据库（如图 3-3 所示）。这两个选择不接受多个数据库参数，所以如果要列出多个数据库，需要重复几次某个选项。

164 例如，要复制除了 top 和 secret 数据库以外的东西，在配置文件中添加以下选项：

```
[mysqld]
...
binlog-ignore-db = top
binlog-ignore-db = secret
```



使用 binlog-*-db 选项过滤事件意味着这两个数据库不会存到二进制日志中，所以崩溃发生时无法使用即时恢复（PITR）进行恢复。因此，如果想过滤复制流，强烈推荐使用 Slave 过滤器，不要用 Master 过滤器。只有你可以承担数据丢失后果时，可以使用 Master 过滤器。

Slave过滤器

Slave 过滤有更多选项。除了可以过滤基于数据库的事件以外，Slave 过滤器还可以过滤单个表，甚至使用通配符过滤一组表。

replicate-wild 规则考虑数据表的全名，包括数据库名和表名。这个选项的匹配模式与 LIKE 字符串比较函数相同，即下画线（_）表示单个字符，百分号（%）表示任意长度的字符串。注意，匹配模式必须过一段时间才合法。也就是说数据库名和表名是独立匹配的，所以单个通配符只能应用到数据库名或表名上。

- replicate-do-db=*db*
 如果当前数据库是 *db*，则执行这个语句。
- replicate-ignore-db=*db*
 如果当前数据库是 *db*，则丢弃这个语句。
- replicate-do-table=*table* 及 replicate-wild-do-table=*db_pattern.tbl_pattern*
 如果该表的名称正在更新为 table，或者符合匹配模式，则执行表的更新。
- replicate-ignore-table=*table* 及 replicate-wild-ignore-table=db_pattern.*tbl_pattern*
 如果该表名称正在更新为 *table*，或者符合匹配模式，则丢弃表的更新。

服务器决定执行过滤规则之前是否要先进行评估，所以，所有事件在过滤之前都会发送给 Slave。

使用过滤将事件分配给Slave

165

那么，Master 过滤和 Slave 过滤相比，好处和缺点分别是什么呢？简单看来，在 Master 上使用 binlog-*-db 而不是 replicate-*-db 选项过滤事件构造数据库，似乎是个很不错的想法。那样，网络就不会有太多负载，这些无用事件的负载已经被 Slave 分担。但是，前面讲过，Master 过滤可能存在以下问题：

- 由于事件从二进制日志中过滤，并且只有一个二进制日志，所以不太可能“切分”改变然后将数据库的不同部分发送到各个不同服务器。
- 二进制日志还用于 PITR（即时恢复），所以如果服务器出现问题，不可能进行完全

恢复。

- 如果必须切分数据，由于二进制日志已经被过滤且不能“不过滤”，那么无法再进行数据切分。

理想的情况是，将 Master 发送的事件过滤，而写入二进制日志的事件不过滤。最好是 Slave 可以控制过滤，决定什么数据将被复制。但这是不可能的，MySQL 5.1 版本必须使用 `replicate-*` 选项过滤事件，即在 Slave 上进行事件的过滤。



后面会继续讨论关于高级过滤特性的实现问题，包括事件处理过程的任意时间点进行事件过滤，以及复杂的过滤逻辑等。

写这本书的时候，将在哪个版本实现这一高级过滤暂时没有定论。

例如，用一个专门的 Slave 存储用户数据，位于 *app* 数据库的 *users* 表和 *profiles* 表中，关闭服务器，向 *my.cnf* 文件添加下面的过滤选项：

```
[mysqld]
...
replicate-wild-do-table=app.users
replicate-wild-do-table=app.profiles
```

如果你担心网络流量（这在长连接（long-haul）网络上十分重要），你可以在 Master 所在机器上再配置一个中继 Slave，如图 5-6 所示（或者在 Master 的相同网段内），其目的是产生 Master 二进制日志的过滤版本。

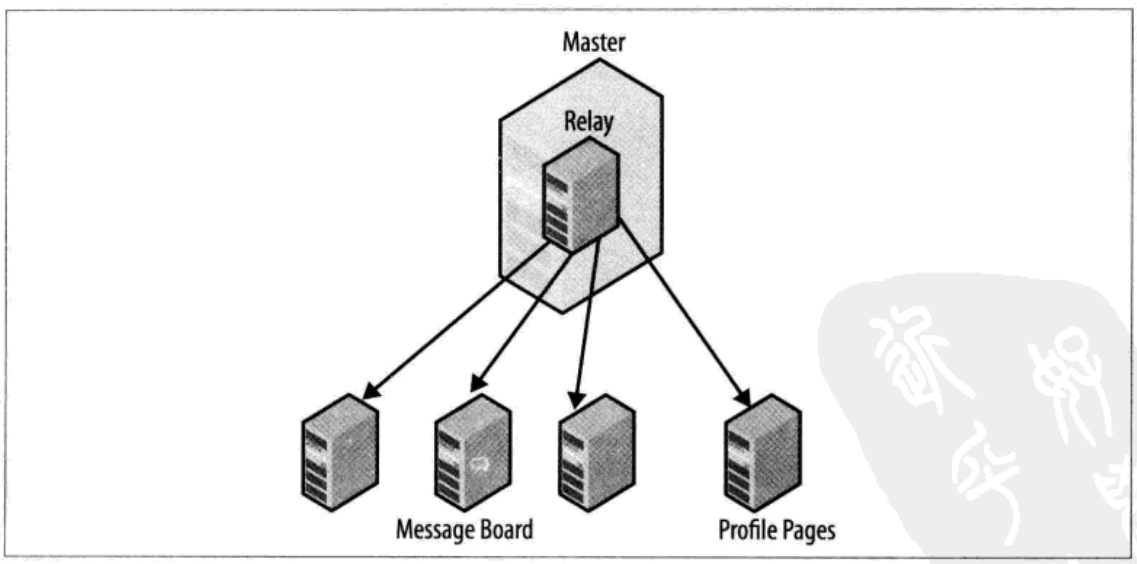


图5-6：将Master和中继服务器放在同一台机器上来过滤

数据分片

你已经知道如何通过向 Master 附加 Slave，并将读操作定向到 Slave 而写操作定向到 Master，来划分（scale）读操作。当负载增加时，很容易添加更多的 Slave 处理更多的读查询操作。所有的写操作仍然都交给 Master 处理，如果写数量增加，Master 可能会成为系统规模增长的瓶颈。有什么方法可以在扩展读操作的同时，同样实现写操作的扩展呢？

在深入讨论扩展写操作的方法之前，先来考虑一下图 5-7 所示的配置，由两个 Master 和一组客户端组成，Master 之间采用双向复制，客户端根据数据变化更新相应的 Master。尽管这个架构看上去似乎处理写操作的能力加倍了（因为这里有两个 Master）但却不能帮助你实现扩展。每个语句都会在两个 Master 上执行，所以写操作的数量也变为双倍的。这并没有改善之前的状况。总之，双主配置（dual-Master setup）并不能扩展写操作，所以必须寻找其他的办法。

图 5-7 中的一对 Master 不能实现扩展的原因是每个语句都要执行两遍。因此，如果语句不是执行两次，就有可能扩展写操作，也就是说，如果服务器之间没有复制，不同服务器之间是完全分离的。

这种架构下，可以通过将数据分割为两个完全不相关的集合来实现写扩展。通过这种方式，客户端试图更新的数据由哪个分区负责，客户端的请求就被定向到哪个分区，从而只需为这个分区的更新处理提供资源，而不管其他分区。

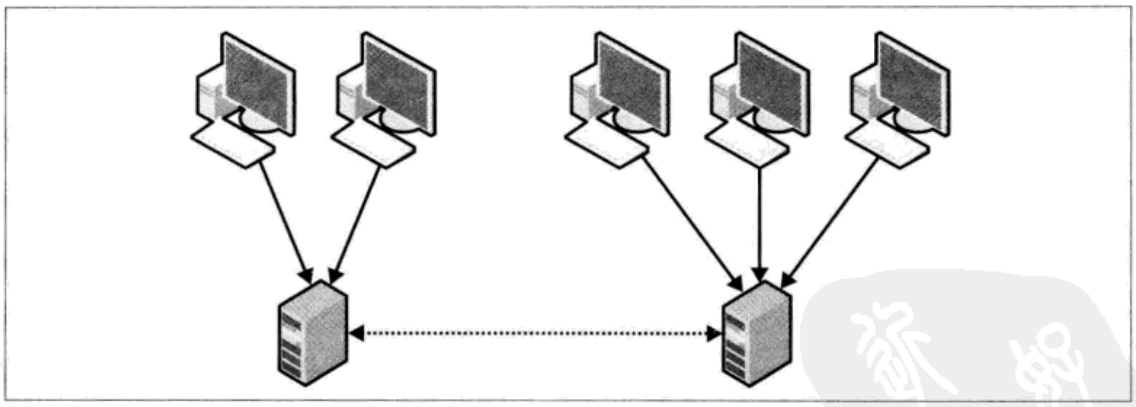


图5-7：双Master双向复制

这种方式的数据分割通常称为分片（*sharding*）（其他常用的名称还有分裂（*splintering*）或水平分割（*horizontal partitioning*），每个分区（*partition*）就是一个分片（*shard*）。通常对大数据分片，如文章、评论、图片和视频，而目录和用户数据则放在中心存储，如

图 5-8 所示。

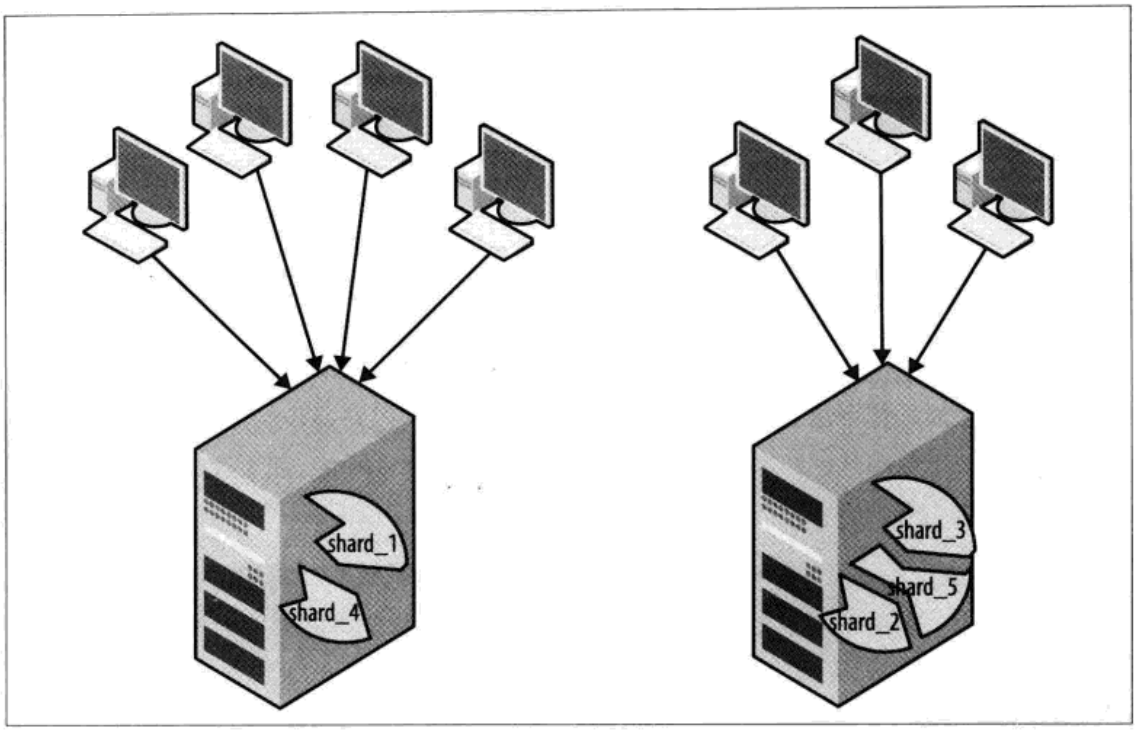


图5-8：几个节点进行分片

168 将相关数据存储在相同节点上可以使查询被定向到单个节点，将分片放在离用户较近的地方也提高了系统的性能，因为这样减少了延迟。将数据分片有以下几个原因：

- 将数据放在离用户距离较近的地方
 前面说过，将大量数据放在离用户近的地方可以减少延迟。
- 减少工作集（*working set*）的大小
 小表的搜索比大表更高效，而且可以进行简单的并行搜索。
 这种方法只有在分片大小基本相同时才有效。所以如果使用这个策略，必须先找到均衡分片大小的方法。
- 负载均衡
 不仅可以通过减小每张表的规模来缩减工作集，分片还可以更加高效地均衡更新负载。如果分片上的更新量很大，可能要将它们分成更小的分片。

数据分片时，可以将多个分片放在同一个服务器上。将几个分片放在同一个服务器上的原因是在系统重新平衡时，将分片移到其他服务器上比较容易，但数据的重新分区却很难。除了能带来管理上的优势以外，较小的分区还使得每张表较小，这也提高了系统的整体性能。

这个架构中，分片的位置不是固定的，需要一种方法将分片 ID 解释为分片所在的节点。通常使用图 5-9 所示的架构来解决这个问题，使用中心存储来跟踪分片的位置。

在已有应用程序中创建分片模式（sharding schema）时，可能一个节点一个分片最简单，但更常见的方式是允许一个节点上存在多个分片。这样可以减少工作集，有希望使查询更快。

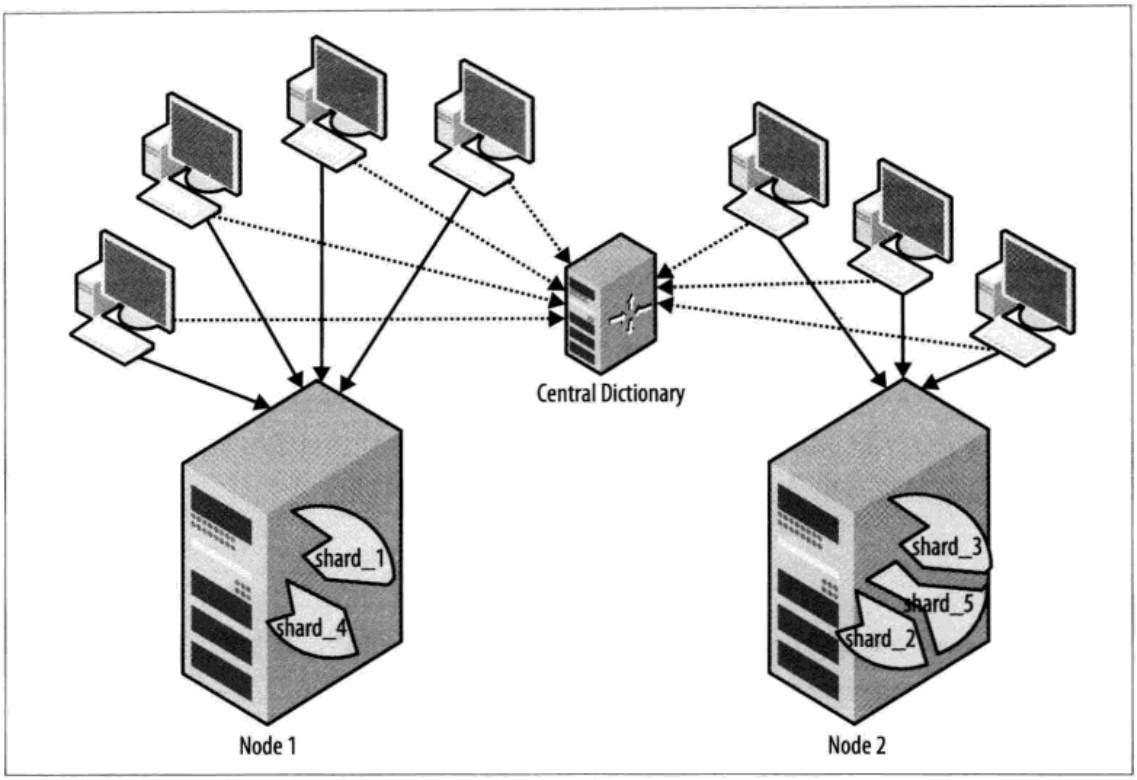


图5-9：中央字典分片

通常分片由应用程序或者数据库层和应用层之间的中间层处理。即使数据库应用层管理分片，也很难在不牺牲性能的情况下完全隐藏分片的结构。让应用知道分片的结构并有效地使用它，这样更好。

MySQL 的分片实现有多种方式，其中常见的两种是 Google 的 Hibernate Shards 和 HiveDB，两者都是 Java 数据库应用层。还可以使用 MySQL Proxy，但它对分片的支持目前还处于起步阶段。MySQL 开发团队将探索有效扩展写操作的解决方案视为重要的任务，其中 MySQL Proxy 可能也是这个方案的一部分。 169

分片的表示

为了使分片结构更有效，需要一种有效的方法表示分片。考虑：

- 很容易移动分片，即在其他地方备份和恢复分片很容易；
- 一个服务器上可能有多个分片，所以需要能够识别同一节点上的不同分片；
- 可以从服务器上复制单个分片来移动这个分片。

满足第一个条件的唯一办法就是将每个分片表示为一个 MySQL 数据库。大多数备份方法可以备份单个数据库，但备份单个表可能较困难。比如，直接复制数据库目录，还有其他备份方法也一样。一个分片一个数据库还能满足最后一个条件，因为可以使用 replicate-do-db 来指定服务器上的某个分片（可以这样实现是因为在 Master 上移动分片，

而且通常 Master 不能从其他地方复制）。

如果一个分片一个数据库，可以在数据库名上附加一个唯一数字来满足第二个条件。也就是说服务器上的分片是名字类似 *shard_123* 的数据库，表的不同分区放在各个数据库中，例如，*posts* 表由 *shard_1.posts*, *shard_2.posts*, …, *shard_N.posts* 组成。

这种方法中，表命名时可以是普通的表名，或者反映其分片的名字。第一种方式可以用 *posts* 表关联 *shard_123.posts*, *shard_124.posts* 等；第二种方式表名为 *shard_123.posts_123*, *shard_124.posts_124* 等。尽管看上去表上的分片数目是多余的，但是当应用程序代码查询了不正确的分片时，这种方法对解决问题很有用。

数据分区

将数据的各个项写入特定服务器允许高效地扩展写操作。但是这对于扩展性来说是不够的：高效地检索数据也很重要，需要将相关数据放在一起。所以，高效分片最大的挑战是选择合适的分区键（*partition key*），使得经常被一起请求的数据存储在同一个分片，或者尽可能少的几个分片上。

创建一个合适的分区键通常需要对数据库的数据结构有些了解。例如，某个应用程序允许专业摄影师安全地存储照片，按照摄影师的位置将照片分片，并将某个摄影师的所有照片放在离他最近的一个分片上。

一旦选择了分区键，下面的问题就是如何使用这个键来分区数据。需要选择一个分区函数，将分区键解释为分片号。两种常见的模式类型如下。

- 静态分片模式

 这种模式下，分区键是静态分配的，一般使用范围或哈希函数。例如，如果分区键是“Country”，则将所有瑞典人放在一个分片上，所有美国人放在另一个分片上（这里我们忽略美国人口是瑞典人口的 30 倍）。

 另一种分片分配基于用户的唯一 ID 字段，使用基于范围的分配，例如第一个分片负责第 0 ~ 9999 个用户，第二个分片负责第 10000 ~ 19999 个用户，依此类推。或

者你也可以使用 ID 的最后四位数字上的哈希值来半随机性地分散用户。

- 动态分片模式 171
 这种模式的分区函数从字典中查找分区键，字典表明了哪个分片包含数据。这种模式比静态模式更加灵活，但是需要一个中心存储来存储字典，这可能会增加查询时间。从高可用性的角度来说，中心存储也是一个问题，因为它会带来单点故障。

你或许已经认识到，静态分片模式在查询分布并不均匀时会出问题——就像前面按国家分片的例子那样。此时，可能美国分片会是瑞典分片的 30 倍。瑞典很乐意这样，假定服务器具有相同的处理能力，他们的响应时间将会很短，而美国访问者的情况则会很糟糕。因此，选择合适的分区键非常重要。

动态分片模式非常灵活。不仅允许更改分片，而且可以在需要的时候轻松实现分片之间的数据移动。灵活性总是需要代价的，为了在正确的分片上检索数据，需要额外的查询分片的操作。使用诸如 Memcached 的缓存系统有些帮助，但最终，好的性能需要与用户的查询模式相匹配的谨慎设计。

创建分区键和分区函数之后，需要将分片号映射到节点（每个服务器上有一个或多个完整的节点）。如果使用静态分片模式，分区函数可以自动将分区号映射到节点。这样很有效率，但以牺牲灵活性为代价，因为移动分片需要重写分区函数。而对于动态分片模式来说，可以实现从分片号到节点的映射，添加一张表来保存这个映射关系。

分片之间的均衡

为了保证即使系统负载发生变化系统仍可响应，或者为了某些管理性原因，有时候需要移动数据，将整个分片移动到其他节点或者在分片之间移动数据。这两个过程都要求保证负载达到重新平衡时宕机时间最小——最好没有宕机时间。解决方案最好是自动的。

将分片移动到其他节点

最简单的方法就是将整个分片移动到其他节点。如果按照前面的建议，将每个分片放在一个单独的数据库中，移动数据库的操作很简单，只需移动字典。但是，移动数据库的同时允许向节点写入则完全不同。

将分片从一个节点（源节点）移动到另一个节点（目标节点），且保证宕机时间尽可能少。 172
这与第 2 章创建 Slave 的技术类似。思想是备份这个分片，在目标节点上恢复，然后使用复制重新执行这个过程中发生的所有改变。

1. 在源节点上创建数据库的备份。为方便练习，假定数据库名为 *shard_123*。联机或脱机备份方法都可以。

2. 我们在上一章中看到的每个备份将备份时间点的数据写入二进制日志中。记下此时日志文件的位置。

3. 停止服务器，关闭目标节点。

4. 在服务器关闭的时候：

 a. 在配置文件中设置 *replicate-do-db* 选项，仅复制需要移动的分片。

   ```
   [mysqld]
   replicate-do-db=shard_123
   ```

 b. 如果要在服务器关闭时恢复源节点的备份，就在这时候完成。

5. 重新启动服务器。

6. 配置复制，从步骤 2 记下的位置开始读取，并在目标服务器上启动复制。从源服务器上读取事件，并将所有变化应用到将要移动的分片上。

 在目标节点上预留额外的处理能力，以应对该节点的写请求数量增加。

7. 当目标节点离源节点足够近时，将源节点上的分片数据库加锁，以阻止改变。不需要阻止目标节点上的分片改变，因为该分片上还没有写请求。

 最简单的方法是发出 LOCK TABLES 命令锁定分片中的所有表，但也有其他方法，比如简单地删除表（如果应用程序可以处理丢失的表，这便是一个可能的替代方案）。

8. 检查源服务器上的日志位置。由于分片不再更新，这将是你需要恢复的日志的最高位置。

9. 等待目标服务器赶上这个位置，例如，使用 START SLAVE UNTIL 和 MASTER_POS_WAIT 语句。

10. 发出 RESET SLAVE 命令关闭目标服务器上的复制。这样会删除所有复制信息，包括 *master.info, relay-log.info* 和所有中继日志文件。如果需要向 *my.cnf* 文件添加其他选项配置复制，则必须删除它们，这在下一步进行。

173

11. 本步骤可选，关闭目标服务器，将目标服务器上的 *my.cnf* 文件的 replicate-do-db 选项删除，然后重新开启服务器。

 这一步不是必需的，因为 replicate-do-db 选项只用来移动分片，且分片移动后功能不受影响。但如果需要再次移动该分片，则需要改动此选项。

12. 更新分片信息，使得更新请求被定向到新的分区位置。

13. 解锁数据库，重新开始分片上的写操作。

14. 删除源服务器上的分片数据库。由于分片锁定的方式不同，可能此时还有对分片的读操作，所以必须要考虑这一点。

哇，有这么多步骤啊。幸运的是，使用 MySQL Replicant 库可以自动完成这些步骤。根据应用程序的实现不同（后面章节讨论示例应用的时候会给出例子），具体每个步骤的细节也会不同。

在分片之间移动数据

即使你现在已经知道了移动分片的策略，有时候还是不得不在分片之间移动某些数据项（data item）。继续上面的照片存储应用的例子，最后用户可能要转移，或者使用模式显示如果照片放在不同的分片上部署会更加高效。想象一下，一个生活在加州但家在纽约的摄影师大部分时候都在查看照片，这就需要能够自动地对系统进行重新分片，并将用户移动到其他分片。

在分片之间移动数据的方法对应用程序的依赖程度很高，所以只能通过例子来说明。通常分片之间移动数据成本很高，因为不得不将分片较长时间离线，以防止移动过程中对用户的改变。

一个分片的例子

为了展示建立分片的方法，我们给出一个小例子，最初使用静态分片模式，然后改为动态分片模式。有一个简单的博客站点，用户注册以后可以发表文章和评论。

本例中，创建一个应用，用户可以发表博文，其他人可以对此文评论。基本数据库的设计如图 5-10 所示，就像你看到的那样，它很简单。

例 5-6 给出了常见数据的定义，为了方便起见将常见数据放在 *common* 数据库中。常见数据只有一个 *user* 表，用来存储所有用户信息，但后面我们会添加更多的表。

例5-6：常见数据的表

```
CREATE TABLE user (
  user_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name CHAR(30),
  password CHAR(30)
);
```

174

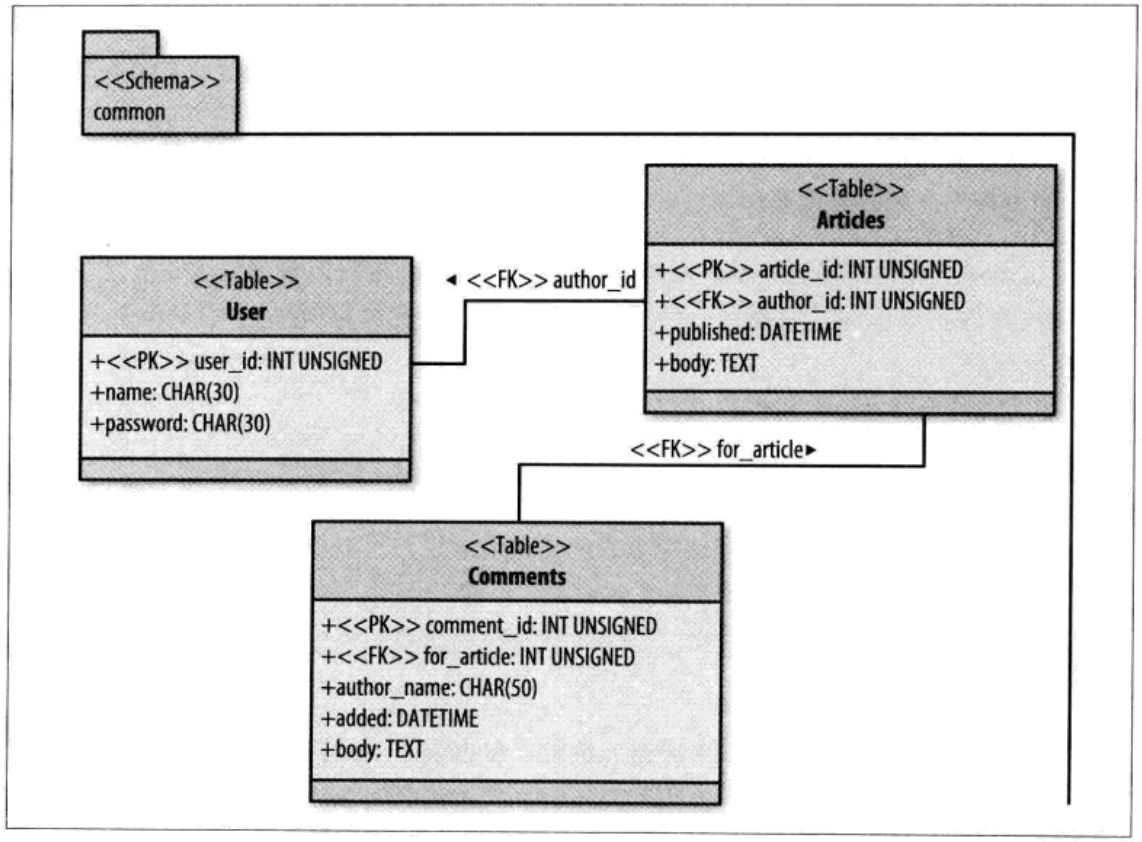


图5-10：数据库设计UML图

我们把文章和评论的表放在另一个数据库中，如例 5-7 所示，*articles* 表存储文章，*comments* 表存储评论。

例5-7：文章及其评论的表

```
CREATE TABLE articles (
  article_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
author_id INT,
  published DATETIME,
175  body TEXT
);
CREATE TABLE comments (
  comment_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  for_article INT,
  author_name CHAR(50),
  added DATETIME,
  body TEXT
);
```

这个应用很简单，但很适合用来描述分片应用。为简单起见，我们故意不设置外键。像这样的集中式数据库，访问频繁且多个客户端并发访问，存储引擎最好选择有行锁的，

比如 InnoDB。

数据库分片

如果用户大量增长，发表的文章数也会大量增加。单个节点上的用户数据管理很简单，所以不需要将 user 表分片——存储 1 亿用户只需 6GB 数据，所以存储空间不是问题，尽管可能有别的原因需要避免表中包含 1 亿条记录。而文章及其评论的数据量可能非常大，因此需要将这些表分片。

文章和评论被分成很多不同的分片，文章的评论与该文章存储在同一个分片上，这样用户就可以用单个查询从同一个分片中高效地检索文章及其评论。我们用数字给分片命名，并将例 5-7 所示的表定义放在每个分片上。为方便起见，分片的基础名就是 *shard*，并加上分片号，例如 *shard_123*。

分区键和分区函数

查找某个用户的应用可能需要列出该用户所发表的所有文章的标题。查找某篇文章常常还想检索该文章的所有评论。

为了高效地满足这些使用模式，我们将数据分片，使得同一个用户的所有文章在同一个分片，某篇文章的所有评论与该文章在同一个分片上。也就是说，本例有多个分区键：

- 如果检索用户，userID 是分区键；
- 如果检索文章，articleID 是分区键。

同样，创建两个分区函数，一个从 userID 到分片号，另一个从 articleID 到分片号。

- 从 userID 到分片号
 简单地对 userID 取模得到分片号。例如，如果有 100 个分片，userID 为 192.，则分片号为 92（192 mod 100）。
- 从 articleID 到分片号
 从 articleID 得到分片号不太容易，因为首先要找到这篇文章的作者。最简单的方法是应用程序在请求文章时，保证代码中有 userID 可用，不过这种方法特定于具体应用。如果某个用户从另一个用户的页面上选择某篇文章，这时很简单（只要添加 HTML 表单作为隐藏字段），但其他情况比较困难。
 另一种方案是为 common 数据库提供需要的信息，引入一个特殊的表，存储 articleID 到 userID 的映射。

176

我们不打算介绍与应用相关的编码，但会简单解释一下如何向 common 数据库中添加表。即使使用静态分片模式（动态分片模式的使用后面再介绍），这种方法比将分片模式编

码到应用代码更加灵活，分片信息的动态更新能力很重要。

例 5-8 给出了一些必要的更改，注意 shard 列上有索引，因为这个列将被频繁查询。

例5-8：定位分片和节点的表

```
CREATE TABLE user (
  user_id INT UNSIGNED AUTO_INCREMENT,
  name CHAR(50), password CHAR(50),
  PRIMARY KEY (user_id)
);
CREATE TABLE node_for (
  shard INT UNSIGNED,
  host  CHAR(28),
  port  INT UNSIGNED,
  sock  CHAR(64),
  KEY (shard)
);
CREATE TABLE user_for (
  article_id INT UNSIGNED,
  user_id INT UNSIGNED,
  PRIMARY KEY (article_id)
);
```

使用实例 5-8 中的表，给定 articleID 或者 userID，很容易定义找到分片号和节点的函数，
177 如例 5-9 所示。每个函数都会发起对拥有常见数据的服务器的 SQL 查询。如果需要检索 userID 及其相应的 articleID，可以使用子查询合并多个查询，如果服务器离得很远，这样可以减少往返时间。

注意单独的 shardNumber 函数。目前这个函数实现了静态分片模式，使用简单的取模函数将 userID 映射到分片号。本章后面将改成使用动态分片模式实现。

例5-9：检索分片号和节点地址的PHP函数

```
  function shardNumber($userId)
  {
    return $userId % 4;
  }

  $NODE = array();
  $NODE[] = array("localhost", 3307, "/var/run/mysqld/mysqld1.sock");
  $NODE[] = array("localhost", 3308, "/var/run/mysqld/mysqld2.sock");
  $NODE[] = array("localhost", 3309, "/var/run/mysqld/mysqld3.sock");
  $NODE[] = array("localhost", 3310, "/var/run/mysqld/mysqld4.sock");

  function getShardAndNodeFromUserId($userId, $common)
  {
    global $NODE;
1   $shardNo = shardNumber($userId);
```

```
2   $row = $NODE[$shardNo % count($NODE)];
    $db_server = $row[0] == "localhost" ? ":{$row[2]}" : "{$row[0]}:{$row[1]}";
    $conn = mysql_connect($db_server, 'query_user');
3   mysql_select_db("shard_$shardNo", $conn);
    return array($shardNo, $conn);
  }

  function getShardAndNodeFromArticleId($articleId, $common) {
    $query = "SELECT user_id FROM article_author WHERE article_id = %d";
    mysql_select_db("common");
    $result = mysql_query(sprintf($query, $articleId), $link);
    $row = mysql_fetch_row($result);
    return getShardAndNodeFromUserId($row[0], $common);
  }
```

更新或读取分片

有了分片号和节点之后，接下来要创建函数从分片中检索信息。例 5-10 定义了两个这样的函数：

- `getArticlesForUser`

 这个函数传入 userID，返回这个用户所发表的所有文章数组。分区函数保证了所有文章都在同一个分片上，所以函数的第一行计算得到所有文章共享的分片号。然后第二行得到分片对应的节点。接着，得到分片上的数据库名（第三行），然后向该节点发送单个查询，检索分片上的所有文章。 178

- `getCommentsForArticle`

 这个函数传入 userID 和 articleID，返回文章及其所有评论的数组。这种特殊情况下，userID 是完整 articleID 的一部分，所以调用者无须进一步查询就可以得到。

这些函数很容易理解，得到正确的分片之后，足以向正确的节点发送查询。由于同一个节点上有多个分片，必须保证读取的是正确的数据库。为简单起见，这里函数不考虑错误处理。

例5-10：检索文章及其评论的PHP函数

```
function getArticlesForUser($userId, $common)
{
  $query = <<<END_OF_SQL
SELECT author_id, article_id, title, published, body
  FROM articles
 WHERE author_id = $userId
END_OF_SQL;
  list($shard, $node) = getShardAndNodeFromUserId($userId, $common);
  $articles = array();
  $result = mysql_query($query, $node);
  while ($obj = mysql_fetch_object($result))
```

```
    $articles[] = $obj;
  return $articles;
}

function getArticleAndComments($userId, $articleId, $common)
{
  list($shard, $node) = getShardAndNodeFromArticleId($articleId, $common);

  $article_query = <<<END_OF_SQL
SELECT author_id, article_id, title, published, body
  FROM articles
 WHERE article_id = $articleId
END_OF_SQL;

  $QUERIES[] = $article_query;
  $result = mysql_query($article_query, $node);
  $article = mysql_fetch_object($result);

  # Fetch the comments from the same shard
  $comment_query = <<<END_OF_SQL
SELECT author_name, body, published
  FROM comments
 WHERE article_ref = $articleId
END_OF_SQL;

  $result = mysql_query($comment_query, $node);
  $comments = array();
179 while ($obj = mysql_fetch_object($result))
    $comments[] = $obj;
  return array($article, $comments);
}
```

本例中直接从分片中读取数据，但如果需要扩展读操作，读请求就会被定向到 Slave。这些就很容易实现了。

实现动态分片模式

目前为止讨论的方法有个缺点，即分区函数是静态的，也就是说如果某个节点流量很大，将分片从一个节点移动到另一个节点并不容易，因为需要改动应用程序代码。

例如，前面用到的简单博客应用。如果某个用户突然发表了一些有趣的文章而有很多人关注，那么他的分片会变得很“热”。这会导致分片之间的不平衡，因为用户出名导致有些分片变热（hot），而不活跃的用户使得其他一些分片变冷（cold）。如果同一个分片上有很多活跃用户，该分片的查询数量可能会增加，直到在可接受的响应时间内回答所有查询变得很困难。解决方法是将用户从热分片移到较冷的分片，但当前的模式无法做到这一点。

动态分片让你的程序在节点之间移动分片以应对流量变化。为此，common 数据库需要做些改动，添加包含分片位置信息的表。新添加的信息放在 *user* 表中最方便。

例 5-11 展示了新加了 *shard_to_node* 表的数据库，该表将分片号映射到对应节点。*User* 表增加了一列表明用户位于哪个分片上。

例5-11：更新动态分片的common数据库

```
CREATE TABLE user (
  user_id INT UNSIGNED AUTO_INCREMENT,
  name CHAR(50), password CHAR(50),
  shard INT UNSIGNED,
  PRIMARY KEY (user_id)
);

CREATE TABLE shard_to_node (
  shard INT UNSIGNED,
  host  CHAR(28),
  port  INT UNSIGNED,
  sock  CHAR(64),
  KEY (shard)
);

CREATE TABLE article_author (
  article_id INT UNSIGNED,
  user_id INT UNSIGNED,
  PRIMARY KEY (article_id)
);
```

180

为了找到分片的节点位置，必须对发出查询的 PHP 函数做出适当修改，从 *shard_to_node* 表提取分片位置，例 5-12 给出了修改之后的 PHP 函数。注意节点数组已经没有了，取而代之的是查询 *common* 数据库中的 *shard_to_node* 表，计算分片号的函数现在改为查询 user 表得到用户所在的分片。

例5-12：使用新的动态分片模式，更改后的PHP函数

```
function shardNumber($userId, $common)
{
  $result = mysql_query("SELECT shard FROM user WHERE user_id = $userId",
$common);
  $row = mysql_fetch_row($result);
  return $row[0];
}

function getShardAndNodeFromUserId($userId, $common)
{
  $shardNo = shardNumber($userId);
  $query = "SELECT host,port,sock FROM shard_to_node WHERE shard = %d";
  mysql_select_db("common", $common);
```

```
  $result = mysql_query(sprintf($query, $shardNo), $common);
  $row = mysql_fetch_row($result);
  $db_server = $row[0] == "localhost" ? ":{$row[2]}" : "{$row[0]}:{$row[1]}";
  $conn = mysql_connect($db_server, 'query_user');
  mysql_select_db("shard_$shardNo", $conn);
  return array($shardNo, $conn);
}
```

我们已经知道如何在动态系统中找到分片了，下一步是添加代码将分片移到新节点或使用新分片。这就是下一节的主题。

分片的再平衡

从静态分片转为动态分片为我们提供了平衡系统的工具，即在节点之间移动分片，以及在分片之间移动数据的能力。这些方法可以作为重新分片解决方案的一部分，即所有分片之间的数据完全再平衡。

好在很容易能将整个分片从一个节点移动到另一个节点。首先创建分片的备份，然后在另一个节点上恢复。如果每个分片都是一个数据库，而且每个数据库都表示为文件系统中的目录，那么移动分片有几种选择。

181



数据库中对象的定义通常存储在文件系统中，但不是所有对象都存在目录中。存储例程和事件的定义是个例外，它们存储在 *mysql* 数据库中，而且根据存储引擎的不同，数据库中的数据可能不需要存在数据库信息目录中。

所以，要检查通过移动目录来移动数据库是否真的移动了所有的对象和数据。

第 12 章将讲述各种备份技术，所以这里就不罗列了。注意在设计一个方案时，不要把过程绑定到任何特定的备份方法，因为可能以后其他备份方法更适合。

为了实现刚刚描述的备份过程，必须用某种技术将分片离线，即阻止对分片的更新。你可以在应用程序中锁定该分片，或者在数据库中锁定表来达到这一目的。

为了防出现未知冲突，在应用程序中实现锁定来协调所有请求，而且由于 Web 应用天生的分布特性，锁的管理会变得相当复杂。

这里简单地锁定一张表(*shard_to_node* 表)，而不是在很多客户端访问的多个表上都加锁。基本上所有分片位置的查询都需要经过 *shard_to_node* 表，所以在这个表上简单地加个锁就保证了在执行分片的移动和重新映射时，分片不会被更新。在此过程中可能有更新正在进行，分片上的更新要么已经启动，要么即将开始。通过锁定分片，正在进行中的更新允许被执行完毕，而即将开始的更新则要等到锁被释放以后。分片上的锁释放以后，原来的分片就没有了，所以更新语句会失败，需要在新的分片上重做。

使用 Replicant 库自动化这个过程（如例 5-13 所示）。

例5-13：在节点之间移动分片的过程

```
_UPDATE_SHARD_MAP = """
UPDATE shard_to_node
   SET host = %s, port = %d, sock = %s
 WHERE shard = %d
"""

_LOCK_SHARD_MAP = """
BEGIN;
SELECT host, port, sock
  FROM shard_to_node
 WHERE shard = %d FOR UPDATE
"""

_UNLOCK_SHARD_MAP = "COMMIT"

def lock_shard(server, shard):
    server.use("common")
    server.sql(_LOCK_SHARD_MAP, (shard))

def unlock_shard(server):
    server.sql(_UNLOCK_SHARD_MAP)

def move_shard(common, shard, source, target, backup_method):
    backup_pos = backup_method.backup_to()
    config = target.fetch_config()
    config.set('replicate-do-db', shard)
    target.stop().replace_config(config).start()
    replicant.change_Master(target, source, backup_pos)
    replicant.Slave_start(target)

    # Wait until Slave is at most 10 seconds behind Master
    replicant.Slave_status_wait_until(target,
                                      'Seconds_Behind_Master', lambda x: x < 10)
    lock_shard(common, shard)
    pos = replicant.fetch_Master_pos(source)
    replicant.Slave_wait_for_pos(target, pos)
    lock_database(target, shard_name)
    common.sql(_UPDATE_SHARD_MAP,
               (target.host, target.port, target.socket, shard))
    unlock_shard(common, shard)
    source.sql("DROP DATABASE shard_%s", (shard))
```

182

前面讲过，即使表被锁定，有些客户端 session 可能正在使用该表，因为它们已经检索到节点位置但尚未连接到节点，或者也可能已经开始更新分片。

应用程序代码必须要考虑这个问题。最简单的解决办法是如果分片上的查询失败，则让

应用程序重新计算节点。我们认为失败意味着分片最近可能被迁移，必须重新查找。例5-14给出了getArticlesForUser函数必须做的改动。

例5-14：处理分片迁移的应用程序代码改动

```
function getArticlesForUser($userId, $common)
{
  global $QUERIES;
  $query = <<<END_OF_SQL
SELECT author_id, article_id, title, published, body
  FROM articles
 WHERE author_id = %d
END_OF_SQL;

  do {
    list($shard, $node) = getShardAndNodeFromUserId($userId, $common);
    $articles = array();
183 $QUERIES[] = sprintf($query, $userId);
    $result = mysql_query(sprintf($query, $userId), $node);
  } while (!$result && mysql_errno($node) == 1146);
  while ($obj = mysql_fetch_object($result))
    $articles[] = $obj;
  return $articles;
}
```

从前面的章节中可以看到，有时候有些用户突然变得很受欢迎，从而单个数据项也需要在分片之间迁移。

迁移用户比迁移分片复杂得多，因为需要抽取分片中的用户及其所有相关的文章和评论，然后在另一个分片上重新安置。这种技术对应用程序的依赖程度很高，所以这里提供的思想仅供参考。

我们将展示一种将用户从源分片迁移到目标分片的技术。这个过程要求数据表中有行锁（例如InnoDB），所以在MyISAM表之间迁移用户的过程对锁的处理将有所不同。相应的Python代码很容易理解，所以我们只关心SQL代码。

如果源分片和目标分片位于同一个节点上，迁移用户很简单，可以通过下面的过程完成。假定数据库中包含它们的分片号。我们用占位符*old*和*new*表示原分片和新分片，用*userID*指代用户。

1. 在common数据库中锁定该用户所在的行，阻塞想要访问该用户的所有session。

```
common> BEGIN;
common> SELECT shard INTO @old_shard
     ->  FROM common.user
     -> WHERE user_id = UserID FOR UPDATE;
```

2. 将用户的文章和评论从原分片迁移到新分片上。

```
shard> BEGIN;
shard> INSERT INTO shard_new.articles
    ->    SELECT * FROM shard_old.articles
    ->     WHERE author_id = UserID
    ->    FOR UPDATE;
shard> INSERT INTO shard_new.comments(comment_id, article_ref, author_name,
    ->                                body, published)
    ->    SELECT comment_id, article_ref, author_name, body, published
    ->      FROM shard_old.comments, shard_old.articles
    ->      WHERE article_id = article_ref AND user_id = UserID;
```

3. 更新用户信息，使其指向新分片。

```
common> UPDATE common.user SET shard = new WHERE user_id = UserID;
common> COMMIT;
```

4. 删除原分片上的用户文章和评论信息。

184

```
shard> DELETE FROM shard_old.comments
    ->       USING shard_old.articles, shard_old.comments
    ->     WHERE article_ref = articles_id AND author_id = UserID;
shard> DELETE FROM shard_old.articles WHERE author_id = UserID;
shard> COMMIT;
```

这种情况下，我们必须打开两个连接：一个用于 common 数据库所在的节点，另一个用于分片所在的节点。如果分片和 common 数据库在同一个节点上，问题就变得非常简单，但我们不能做这样的假设。

如果分片位于不同的数据库，下面的过程可以较为简单地解决问题。

1. 在源节点上创建文章和评论的备份，同时，得到备份对应的 binlog 位置。

为此，锁定 *articles* 表和 *comments* 表中该用户相关的行。注意，需要启动事务来完成，类似于迁移分片时更新 *shard_to_node* 表的事务，这样足以阻塞写操作，但无法阻塞读操作。

```
shard_old> BEGIN;
shard_old> SELECT * FROM articles, comments
        ->  WHERE article_ref = article_id AND author_id = UserID
        ->    FOR UPDATE;
```

2. 创建文章和评论的备份。

```
shard_old> SELECT * INTO OUTFILE 'UserID-articles.txt' FROM articles
        ->  WHERE author_id = UserID;
shard_old> SELECT * INTO OUTFILE 'UserID-comments.txt' FROM comments
```

```
    ->  WHERE article_ref = article_id AND author_id = UserID;
```

3. 将保存的文章和评论复制到新节点上，然后使用 LOAD DATA INFILE 写入新节点。

```
shard_new> LOAD DATA INFILE 'UserID-articles.txt' INTO articles;
shard_new> LOAD DATA INFILE 'UserID-comments.txt' INTO comments;
```

4. 更新 common 数据库中用户的分片信息。

```
common> UPDATE user SET shard = new WHERE user_id = UserID;
```

5. 在原分片上删除用户的文章和评论信息，这一步与前面的过程一样。

```
shard_old> DELETE FROM comments USING articles, comments
        ->     WHERE article_ref = articles_id AND author_id = UserID;
shard_old> DELETE FROM articles WHERE author_id = UserID;
shard_old> COMMIT;
```

185

数据的一致性管理

本章前面讨论过，异步复制的问题之一就是一致性管理。为了说明这个问题，设想你有一个电子商务站点，客户浏览找到他们想购买的东西，然后放入购物车。你有一个服务器集群，用户向购物车中添加产品时，这种变化请求转发给 Master，而 web 服务器需要知道购物车中的内容信息，这个查询将转发给负责此类查询的 Slave。由于 Master 比 Slave 超前，变化可能还没有到达 Slave，所以转发到 Slave 的查询可能发现购物车为空。这样客户当然会大吃一惊，马上将选中的项目再次加入购物车，最后发现购物车中有两个项目，因为这次 Slave 赶上了 Master，复制了购物车的两次变化。显然要避免这种情况发生，否则可能引起大量客户的不满。

为了避免数据过于陈旧，要保证 Slave 提供的数据是最近的。如果添加中继服务器，问题会变得更加棘手。解决的基本思想是标记每个在 Master 上提交的事务，然后在 Slave 执行查询之前等待事务到达 Slave。根据 Master 和 Slave 之间是否有中继 Slave，问题的处理方法也不同。

非级联部署中的一致性

如果所有 Slave 直接连接到 Master，则一致性检查很容易。这时，只要在事务提交后记录 binlog 位置，然后等待 Slave 达到这个位置（使用前面介绍的 MASTER_POS_WAIT 函数）。但是，事务写入 binlog 的确切位置却无法达到。为什么呢？因为在事务提交以后、SHOW MASTER STATUS 执行之前的某个时刻，仍有事件被写入 binlog。

这并不重要，因为并不需要达到事务写入 binlog 的确切位置，只要能到达事务位置或其

后某个位置就够了。SHOW MASTER STATUS 命令显示当前复制正在写的事件位置，事务提交后执行足以获得 binlog 位置，以进行一致性检查。

例 5-15 给出了更新处理的 PHP 代码，保证了展现的数据并不过时。

例5-15：避免读取过时数据的PHP代码 186

```
function fetch_Master_pos($server) {
  $result = $server->query('SHOW MASTER STATUS');
  if ($result == NULL)
    return NULL;                                  // Execution failed
  $row = $result->fetch_assoc();
  if ($row == NULL)
    return NULL;                                  // No binlog enabled
  $pos = array($row['File'], $row['Position']);
  $result->close();
  return $pos;
}

function sync_with_Master($Master, $Slave) {
  $pos = fetch_Master_pos($Master);
  if ($pos == NULL)
    return FALSE;
  if (!wait_for_pos($Slave, $pos[0], $pos[1]))
    return FALSE;
  return TRUE;
}

function wait_for_pos($server, $file, $pos) {
  $result = $server->query("SELECT MASTER_POS_WAIT('$file', $pos)");
  if ($result == NULL)
    return FALSE;                                 // Execution failed
  $row = $result->fetch_row();
  if ($row == NULL)
    return FALSE;                                 // Empty result set ?!
  if ($row[0] == NULL || $row[0] < 0)
    return FALSE;                                 // Sync failed
  $result->close();
  return TRUE;
}

function commit_and_sync($Master, $Slave) {
  if ($Master->commit()) {
    if (!sync_with_Master($Master, $Slave))
      return NULL;                    // Synchronization failed
    return TRUE;                      // Commit and sync succeeded
  }
  return FALSE;                       // Commit failed (no sync done)
}
```

```
function start_trans($server) {
  $server->autocommit(FALSE);
}
```

在例 5-15 中可以看到有 commit_and_sync 函数、start_trans 函数及三个辅助函数，即 fetch_Master_pos，wait_for_pos 和 sync_with_Master。commit_and_sync 函数提交事务，并等待事务到达指定的 Slave。它有两个输入参数，即连接到 Master 的对象和连接到
187 Slave 的对象。如果提交和同步完成，函数就返回 TRUE，提交失败则返回 FALSE，提交成功但同步失败则返回 NULL（可能是 Slave 出错或者 Slave 无法连接上 Master）。

函数提交当前事务以后，如果提交成功，则使用 SHOW MASTER STATUS 命令取得当前 Master 的 binlog 位置。由于在事务提交之后、SHOW MASTER STATUS 命令发出之前，可能已经有其他线程执行了数据库的更新操作，所以可能（很有可能）不是在事务结束的时候返回 binlog 位置，而是在事务写入 binlog 的某个时候。前面提过，从正确性的角度来看，这没什么问题，因为事务总会执行到后来的位置。

从 Master 取得 binlog 位置以后，接下来该函数连接到 Slave，并使用 MASTER_POS_WAIT 函数等待 Master 位置。如果 Slave 正在运行，则阻塞此函数的调用，直到读到这个位置；但如果 Slave 不是正在运行，立即返回 NULL。如果函数正在等待时 Slave 停止运行也会发生这种情况，比如 Slave 线程执行语句时出错。无论哪种情况，NULL 都是指事务尚未到达 Slave，所以检查函数调用的结果很重要。如果 MASTER_POS_WAIT 函数返回 0，则说明 Slave 已经收到这个事务了，因此同步较容易成功。

首先像往常一样连接服务器即可，不过接下来要使用函数来启动、提交和中止事务。例 5-16 给出了函数使用的示例，但这里不考虑错误检查，因为这依赖于错误处理的方式。

例5-16：start_trans 和commit_and_sync函数的使用

```
require_once './database.inc';

start_trans($Master);
$Master->query('INSERT INTO t1 SELECT 2*a FROM t1');
commit_and_sync($Master, $Slave);
```

级联部署中的一致性

级联部署中的一致性管理与简单复制拓扑结构中的一致性管理大不相同，简单复制拓扑中每个 Slave 都直接连接到 Master。这时候等待 Master 位置是不可能的，因为每个中间的中继服务器都会改变这个位置。要想想其他办法来等待事务。说到处理等待，MASTER_POS_WAIT 函数很适合，所以如果使用这个函数，可以解决很多问题。大体上有

两种方案来保证读取到的数据没有过时。

第一种是方案依靠全局事务 ID（在第 4 章中介绍过）来处理 Slave 提升，并反复轮询 Slave 直到 Slave 处理这个事务。 188

第二种方案是在 Master 到最后一个 Slave 的路径上连接所有的中继服务器，保证所有的变化都能到达到 Slave，如图 5-11 所示。Master 和 Slave 之间的每个中继 Slave 都要连接，因为无法知道各个中继服务器会使用哪个 binlog 位置。

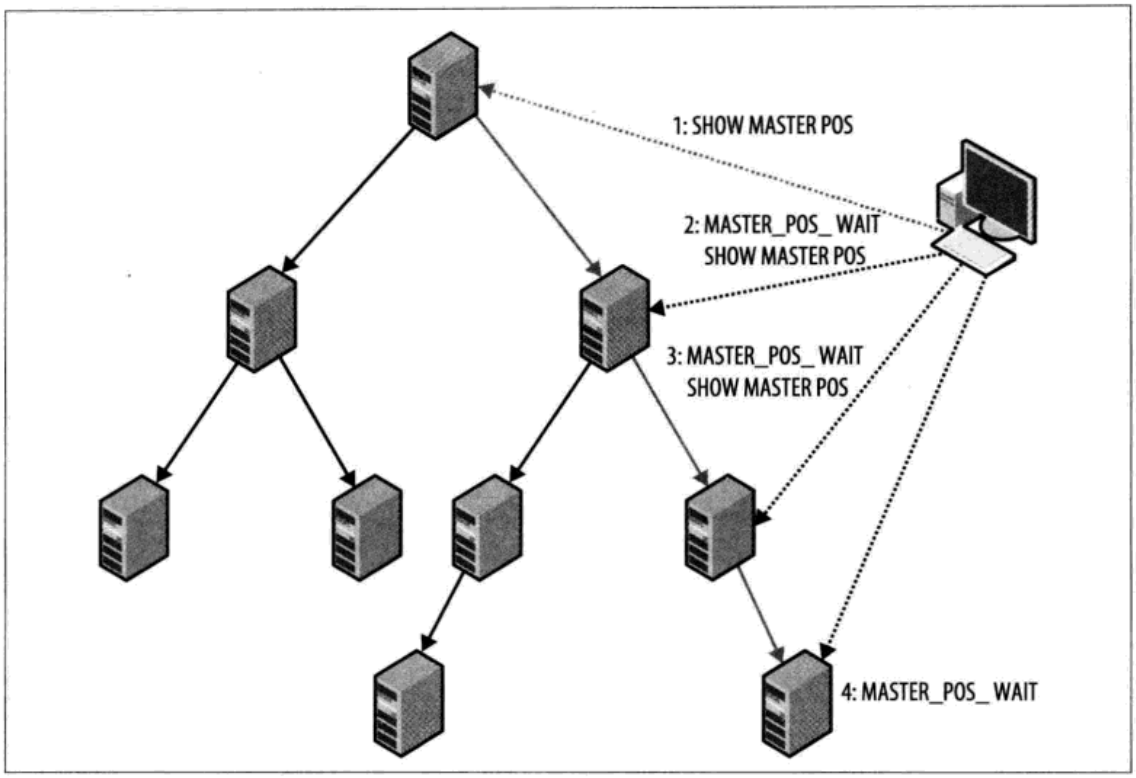


图5-11：在中继链上同步所有服务器

两种方案各有长处，下面来看看它们各自的优缺点。

如果 Slave 一般保持相对于 Master 的新数据，第一种解决方案是仅对最终 Slave（final Slave）进行简单的检查，通常事务已经被复制到 Slave，处理过程可以继续。如果事务还没有处理，可能在下一次检查之前处理，所以在最终 Slave 第二次被检查时，事务就会到达 Slave。如果检查的时间间隔很短，用户就不会察觉到延迟，所以典型的一致性检查在轮询最终 Slave 时需要一个或多个额外的消息。这种方法只需要轮询最终 Slave，而不用轮询中间 Slave。从管理的角度来看，这也是个优势，因为不需要跟踪中间 Slave，也不用关心它们之间是如何连接的。 189

另一方面，如果 Slave 滞后（lag behind），或者复制延迟各不相同，此时使用第二种方案会更好。第一个方法反复轮询 Slave，而大部分时候事务并没有在 Slave 上提交。可以增加轮询的时间间隔，但是如果轮询间隔太大，响应时间将变得不可接受，这时第一个方案将不再适用。这种情况下，最好使用第二种方案，等待改变在复制树（replication tree）上扩散，然后执行查询。

一个大小为 N 的树，需要的额外请求的数目与 $logN$ 成正比。例如，如果有 50 个中继服务器，每个中继服务器处理 50 个最终 Slave，那么可以处理 2500 个最终 Slave，并且需要两个额外的请求：一个发给中继 Slave，另一个发给最终 Slave。

第二种方法的缺点是：

- 需要应用程序代码访问中继服务器，从而轮流连接到每个中继服务器，等待并达到指定位置。
- 需要应用程序代码了解复制的架构，以便查询中继服务器。

查询中继 Slave 可能会降低速度，因为它们必须处理更多的事情，但实际上，这或许并不是问题。通过引入缓存数据库连接层，可以避免部分流量问题。每次请求来临，缓存层会记住 binlog 的位置，只有 binlog 位置比缓存位置大时，才查询中继服务器。下面是缓存层粗略的 stub 函数。

```
function wait_for_pos($server, $wait_for_pos) {
  if (cached position for $server > $wait_for_pos)
    return TRUE;
  else {
    code to wait for position and update cache
  }
}
```

由于 binlog 位置总是不断增长的（如果某个 binlog 位置已经达到，这个位置在 binlog 中就处于已达到的状态），因此没有返回不正确结果的风险。要确定哪种技术可以更有效地监控和配置部署，唯一的方法就是确保对应用程序来说执行查询的速度足够快。

例 5-17 给出了第一种方案的代码示例——反复查询 Slave，检查事务是否执行。这个代码使用了第 4 章中提到的 *Last_Exec_Trans* 表，在 Master 上检查该表，然后在 Slave 上反复读取该表，直到发现正确的事务为止。

190

例5-17：使用轮询避免读取过时数据的PHP代码

```
function fetch_trans_id($server) {
  $result = $server->query('SELECT server_id, trans_id FROM Last_Exec_Trans');
  if ($result == NULL)
    return NULL;                                  // Execution failed
```

```
  $row = $result->fetch_assoc();
  if ($row == NULL)
    return NULL;                                  // Empty table !?
  $gid = array($row['server_id'], $row['trans_id']);
  $result->close();
  return $gid;
}

function wait_for_trans_id($server, $server_id, $trans_id) {
  if ($server_id == NULL || $trans_id == NULL)
    return TRUE;        // No transactions executed, trivially in sync

  $server->autocommit(TRUE);
  $gid = fetch_trans_id($server);
  if ($gid == NULL)
    return FALSE;
  list($current_server_id, $current_trans_id) = $gid;
  while ($current_server_id != $server_id || $current_trans_id < $trans_id) {
    usleep(500000);                                  // Wait half a second
    $gid = fetch_trans_id($server);
    if ($gid == NULL)
      return FALSE;
    list($current_server_id, $current_trans_id) = $gid;
  }
  return TRUE;
}

function commit_and_sync($Master, $Slave) {
  if ($Master->commit()) {
    $gid = fetch_trans_id($Master);
    if ($gid == NULL)
      return NULL;
    if (!wait_for_trans_id($Slave, $gid[0], $gid[1]))
      return NULL;
    return TRUE;
  }
  return FALSE;
}

function start_trans($server) {
  $server->autocommit(FALSE);
}
```

commit_and_sync 和 start_trans 函数与例 5-15 和例 5-16 类似，区别在于例 5-17 中的函数内部调用了 fetch_trans_id 和 wait_for_trans_id，而不是 fetch_Master_pos 和 wait_for_pos。代码中有几点需要注意：

- 开始查询 Slave 之前，wait_for_trans_id 中关闭了自动提交选项。这一步是必需的，因为如果是隔离等级是可重复读（repeatable read）或者更高，则每次 SELECT 语 191

句的全局事务 ID 都一样。

- 为了防止上面的情况，开启自动提交选项，在独立的事务中提交 SELECT 语句。或者使用读已提交（read committed）隔离等级。
- 为了避免 wait_for_trans_id 中不必要的 sleep，获取全局事务 ID 并在进入循环之前检查全局事务 ID。
- 本代码只需要访问 Master 和 Slave，而不用访问中间的中继服务器。

例 5-18 给出了确保不读取过时数据的代码，查询 Master 和最终 Slave 之间的所有服务器。这种方法首先查找最终 Slave 和 Master 之间的整条服务器链，然后顺着该链依次同步每个服务器，直到事务到达最终 Slave。代码中重用了例 5-13 中的 fetch_Master_pos 和 wait_for_pos，所以这里不再重复。本代码没有实现任何缓存层的功能。

例5-18：通过等待避免读取过时数据的PHP代码

```
function fetch_relay_chain($Master, $final) {
  $servers = array();
  $server = $final;
  while ($server !== $Master) {
    $server = get_Master_for($server);
    $servers[] = $server;
  }
  $servers[] = $Master;
  return $servers;
}

function commit_and_sync($Master, $Slave) {
  if ($Master->commit()) {
    $server = fetch_relay_chain($Master, $Slave);
    for ($i = sizeof($server) - 1; $i > 1 ; --$i) {
      if (!sync_with_Master($server[$i], $server[$i-1]))
        return NULL;                          // Synchronization failed
    }
  }
}

function start_trans($server) {
  $server->autocommit(FALSE);
}
```

fetch_relay_chain 函数用来查找 Master 和 Slave 之间的所有服务器。从 Slave 出发，
192 使用 get_Master_for 函数到达响应的 Master。本代码中故意没有给出这个函数，因为这对我们的讨论没有任何意义。然而，这个函数还是必须要有定义的。

取得中继链以后，沿着该链同步 Master 及其 Slave。这是通过 sync_with_Master 函数实现的，在例 5-15 中也有这个函数。



获取服务器的 Master 方法是使用 SHOW SLAVE STATUS，然后读取 Master_Host 和 Master_Port 项。但是，如果每个即将提交的事务都这么做，系统将会变得很慢。

因为拓扑结构几乎不变，最好将信息缓存在应用服务器上或者其他地方，这样可以避免数据库服务器的流量过大。

在第 4 章中已经学习了如何处理 Master 故障，例如，将故障转移到另一个 Master 上，或者将 Slave 提升为 Master。而且，一旦 Master 修复以后，需要将其重新部署到现有的环境中。Master 是部署中的关键组件，常常是比 Slave 更加强大的机器，所以在重新部署时要将其恢复到 Master 位置。由于 Master 在没有预期的情况下停止，它很可能与部署中的其他组件不同步。这种情况有两种场景：

- 如果 Master 离线时间较长，系统的其他部分已经提交了很多事务，而 Master 不知道。从某种意义上来说，跟系统的其他部分相比，Master 面临着另一种未来。图 5-12 给出了这种情况的示意图。

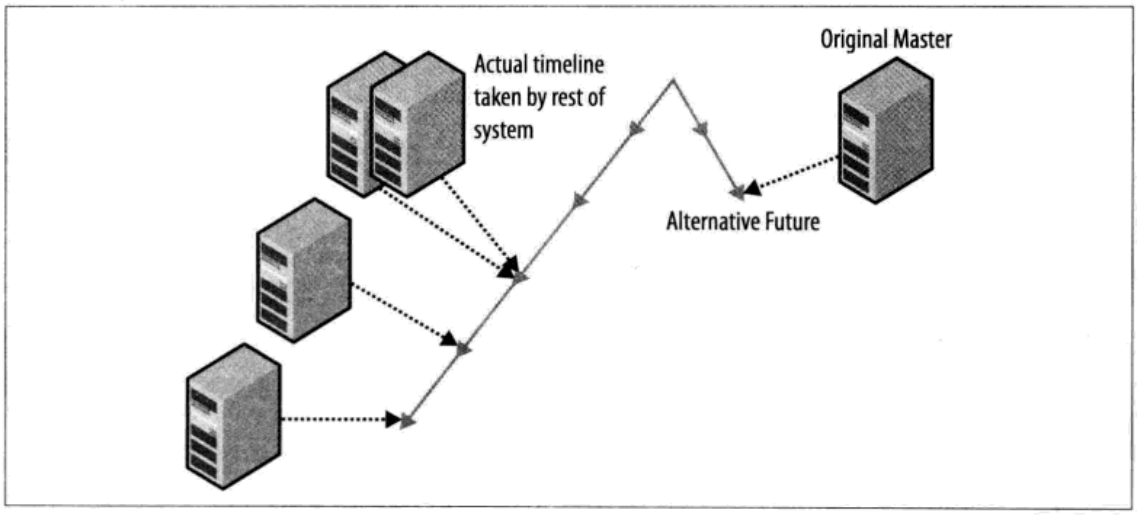


图5-12：原Master的另一种未来

- 如果 Master 提交了事务，并写入二进制日志，然后在确认事务之后崩溃，那么事务可能没有到达 Slave。这意味着 Master 有事务没有被 Slave 处理，系统的其他部分也不知道这些事务。

如果原来的 Master 不是远远落后于当前的 Master，解决第一个问题的最简单方法就是将原来的 Master 作为 Slave 连接到当前的 Master，一旦同步以后将所有 Slave 切换到 Master。但是，如果原来的 Master 已经离线时间较长，克隆其中一个 Slave 可能更快，然后将所有 Slave 切换到 Master。

如果 Master 面临着另一种未来，那么它的额外事务可能不应该带回部署中。为什么呢？因为突如其来的新事务可能与现有的事务存在微妙的冲突。例如，如果事务是消息板上的一条消息，用户可能会重复提交这个消息。如果一个消息已经向消息板写过但却被报
193 告丢失（因为在消息发给 Slave 之前 Master 崩溃了），突然又重新出现，这会使用户迷惑不解，肯定会令人讨厌。同样，因为 Master 修复后被重新部署回系统中，购物车中突然增加的项目也会让用户很不高兴。

简而言之，这两种不同步的问题都可以解决（Master 处于另一种未来和 Master 需要同步的情况），简单地克隆原来 Master 的 Slave，然后依次将每个当前 Slave 切换到原来的 Master 上即可。

但是，这些问题突出了保证一致性是多么重要，倘若 Master 崩溃，在事务完成之前，检查 Master 上的变化在其他系统是否可用。本章中所讨论的代码都假设用户立即读取数据，因此在服务器执行读查询之前检查查询是否达到 Slave。从恢复的角度看，这样做有点过度：保证事务在至少一个其他机器上可用就够了，例如连接到 Master 的其中一个 Master 或者中继服务器。如果 n 个服务器上发生改变，通常可以容忍 n-1 次故障。

小结

本章学习了通过横向扩展来提高应用程序吞吐量的技术，引入了更多的服务器来处理更多的数据请求。还介绍了利用复制来建立 MySQL 横向扩展的方法，并给出了一些概念的具体示例。下一章中将了解更多的高级复制概念。

194 一阵敲门声使 Joel 注意到 Summerson 先生正站在他办公室门前。

“我喜欢你服务器横向扩展的报告，Joel，我希望你马上开始着手做。可以用我们机房中多余的服务器。”

Joel 很高兴当初决定向老板提出建议。“好的，先生。什么时候上线？”

Summerson 先生笑了，瞥了一眼表，说“现在还不是时候。”然后就走了。

Joel 不知道他是不是在开玩笑，所以他决定马上开始。他拿起那本“MySQL High Availability”的复印本和笔记，向机房走去。“希望我设置了 TiV。，”他嘀咕着，他明白这次又要熬夜了。

第6章 高级复制

Joel 正在看邮件，一阵敲门声转移了他的注意力。原来是 Summerson 先生站在门口，Joel 一点儿也不觉得奇怪。

“什么事，先生？”

“我有点担心我们现在所做的复制。我希望你做点研究，看看怎样才能提高对它的理解。我希望你做个文档，不仅要解释当前的配置，还能就出错的情况提出一些排除故障的具体想法，以及怎样让系统运转起来。”

Joel 正想有个这样的任务。他也开始注意到需要了解更多复制的相关知识。

“我马上就去做，先生。”

“很好，这个不急。希望你能够做好。”

Joel 点点头，然后老板离开了。他叹了口气，然后翻出了他所有心爱的 MySQL 方面的书。他需要更加详细地看一下复制的知识。

前面的章节介绍了配置和部署复制的基本知识，以及如何确保站点不宕机运转且可用，但要想理解复制的潜在缺陷，以及如何高效地使用复制，还需要了解其操作的相关知识和完成任务所需的各种信息。这就是本章的目标，内容包括以下几个方面：

- 如何更加健壮地将 Slave 升级为 Master；
- 崩溃后避免数据库损坏的建议；
- 多源复制（Multisource replication）；

- 基于行的复制（Row-based replication）。

196

复制架构基础

第 3 章讨论了二进制日志及审查其中记录的事件的相关工具。但我们没有描述事件是如何发送给 Slave 并在那里重新执行的。一旦你理解了这些细节，就可以更大程度地控制复制，防止崩溃后的损坏，并通过检查日志来审查问题。

图 6-1 展示了内部复制架构的示意图，其中包括连接到 Master 的多个客户端、Master 本身，以及若干 Slave。对每个连接到 Master 的客户端来说，服务器将运行一个会话负责执行所有的 SQL 语句，并将结果返回给客户端。

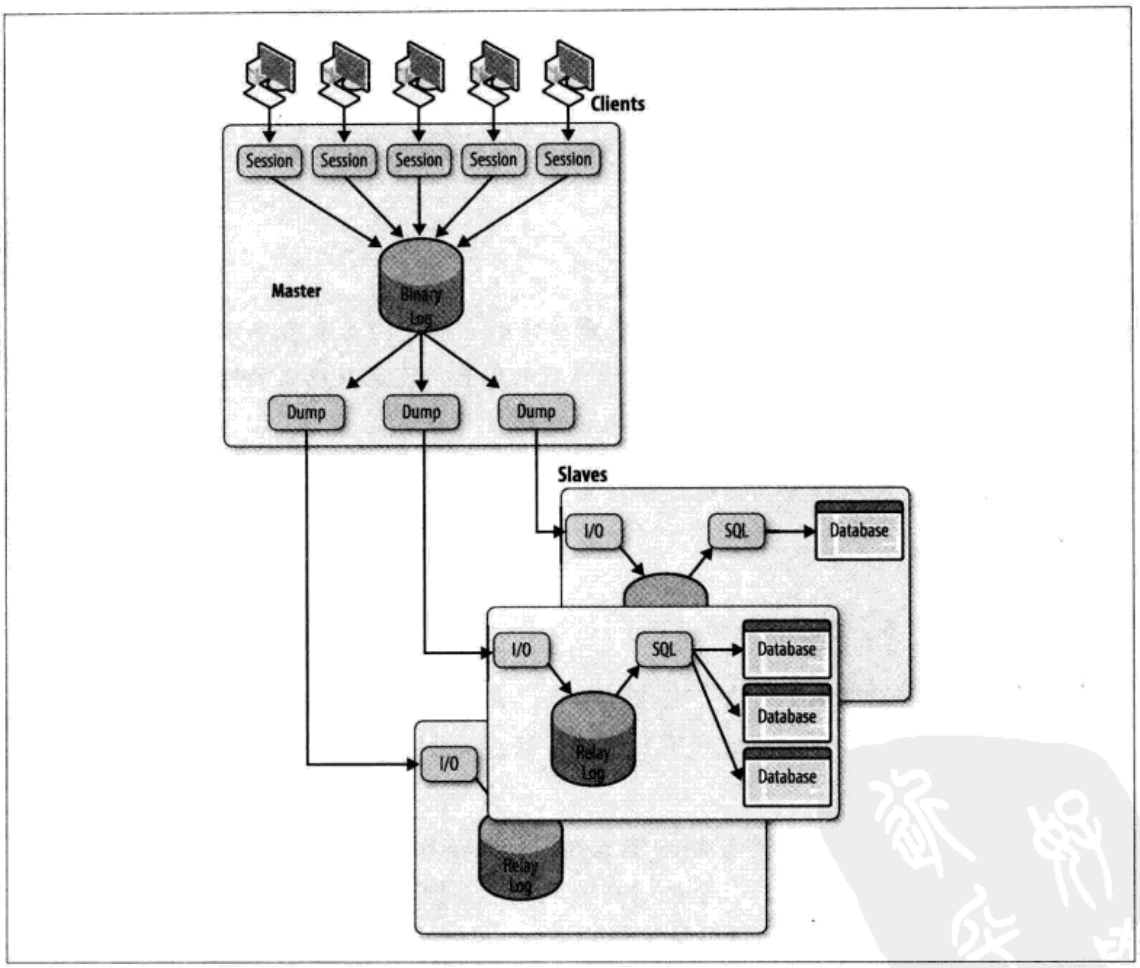


图6-1：Mater和若干Slave的内部结构

复制系统中，Master 到 Slave 的事件流如下：

1. 会话接受来自客户端的语句然后执行，并与其他会话保持同步，保证每个事务都被执行，且不与其他会话产生的改变发生冲突。

2. 语句执行结束之前，向二进制日志中写入一条记录，该记录包含一个或多个事件。第 2 章已经介绍了这个过程，本章不再赘述。

3. 事件写入二进制日志后，Master 的转储线程（*dump thread*）从二进制日志中读取事件，然后将它们发送给 Slave 的 I/O 线程。

4. 当 Slave 的 I/O 线程接收到该事件时，将它写入中继日志（relay log）的末尾。

5. 写入中继日志后，*Slave* 的 *SQL* 线程从中继日志中读取事件并执行，从而在 Slave 的数据库上应用（apply）这些改变。

如果丢失了 Master 连接，Slave 的 I/O 线程将试图重新连接服务器，就像其他 MySQL 客户端线程一样。本章我们将看到有些选项可以处理重新连接问题。

中继日志的结构

197

前面已经知道，中继日志是连接 Master 和 Slave 的信息——它是复制的核心。明白它是如何使用的，以及如何通过它来协调 Slave 线程很重要。因此，这里将深入研究中继日志的结构，以及 Slave 线程是如何使用中继日志来处理复制的。

前面讲过，I/O 线程将来自 Master 的事件存储到中继日志中。将中继日志作为缓冲，这样 Master 不必等待 Slave 执行完成就可以发送下一个事件。

图 6-2 给出了中继日志的示意图，其结构与 Master 的 binlog 类似，不过还包含了一些额外的文件。

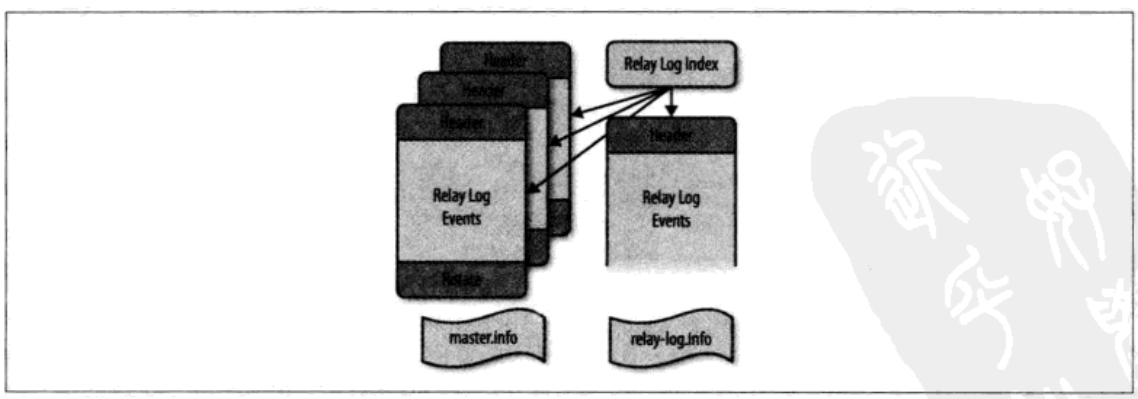


图6-2：中继日志的示意图

198 除了二进制日志中的内容文件和索引文件以外，中继日志还维护两个文件来跟踪复制进度：中继日志信息文件和 Master 日志信息文件。这两个文件的名字由 *my.cnf* 配置文件中的两个参数来控制：

- `relay-log-info-file=`*`filename`*
 这个参数设置了中继日志信息文件名。也可以使用只读服务器变量 `relay_log_info_file`。除非给出完整的文件名，否则文件名与服务器的数据目录相关。默认文件名为 *relay-log.info*。
- `master-info-file=`*`filename`*
 这个参数设置了 Master 日志信息文件名。默认文件名为 *master.info*。



master.info 文件中的信息优先于 *my.cnf* 文件。也就是说，如果改变了 *my.cnf* 文件中的信息，然后重启服务器，将会从 *master.info* 而不是 *my.cnf* 文件中读取信息。

因此，不推荐将 `CHANGE MASTER TO` 命令的参数放在 *my.cnf* 文件中，而是使用 `CHANGE MASTER TO` 命令直接配置。如果由于某种原因，需要把复制参数放到 *my.cnf* 文件，并且保证 Slave 启动时读取这些参数，则必须在编辑 *my.cnf* 文件之前发出 `RESET SLAVE` 命令。

请谨慎执行 `RESET SLAVE`！这个命令会删除 *master.info* 和 *relay-log.info* 文件，以及所有中继日志文件！

199 为方便起见，下面的讨论中我们均使用信息文件的默认文件名。

master.info 文件包含 Master 读取位置，以及连接 Master 和启动复制必需的所有信息。当 Slave I/O 线程启动时，如果 *master.info* 文件可用，线程将从中读取信息。

例 6-1 给出了 *master.info* 文件的简单示例。我们在每行前面了加了行号，且每行末都有斜体注释（文件本身没有注释）。如果服务器不支持 SSL 的编译，则去掉第 9 ～ 15 行——其中包含了所有 SSL 参数。例 6-1 表明了 SSL 编译时这些参数是如何设置的。SSL 字段将在本章后面介绍。



master.info 文件中的密码是没有加密的。因此，文件的保护很重要，要保证只有 MySQL 服务器才能读取它。通常的方法是在服务器上定义一个专门的用户来运行服务器，将负责复制和数据库维护的所有文件都分配给这个用户，并且只分配读写权限，删除其他文件访问权限。

例6-1：master.info文件的内容（支持SSL的MySQL 5.16.版本）

```
1  15                          Number of lines in the file
2  master1-bin.000032          Current binlog file being read (Master_Log_File)
```

```
3  475774                        Last binlog position read (Read_Master_Log_Pos)
4  master1.example.com           Master host connected to (Master_Host)
5  repl_user                     Replication user (Master_User)
6  xyzzy                         Replication password
7  3306                          Master port used (Master_Port)
8  1                             Number of times slave will try to  reconnect
                                 (Connect_Retry)
9  1                             1 if SSL is enabled, otherwise 0
10                               SSL Certification Authority (CA)
11 /etc/ssl/certs                SSL CA Path
12 /etc/ssl/certs/slave.pem      SSL Certificate
13                               SSL Cipher
14 /etc/ssl/private/slave.key    SSL Key
15 0                             SSL Verify Server Certificate (5.1.16 and later)
```



如果是老版本的服务器，格式可能略有不同。

MySQL 4.1 之前的版本没有第一行。开发人员在 4.1.1 版本中添加了行号，这样可以为文件扩展新字段，并通过检查行号来检测支持哪些字段。

最后一行提到了 5.1.16 版本，*SSL Verify Server Certificate*。

relay-log.info 文件跟踪复制的进度，并由 SQL 线程负责更新。例 6-2 给出了 *relay-log.info* 文件节选。这些行对应于将要执行的下一个事件的开始。

例6-2：relay-log.info文件的内容

200

```
./slave-relay-bin.000003     Relay log file (Relay_Log_File)
380                          Relay log position (Relay_Log_Pos)
master1-bin.000001           Master log file (Relay_Master_Log_File)
234                          Master log position (Exec_Master_Log_Pos)
```

如果某个文件不可用，则根据 *my.cnf* 文件中的信息重建，并在 Slave 启动时，将参数传递给 `CHANGE MASTER TO` 命令。



仅仅使用 *my.cnf* 文件配置 Slave 并执行 `CHANGE MASTER TO` 命令是不够的。只有执行了 `START SLAVE` 命令，中继日志文件、*master.info* 文件和 *relay-log.info* 文件才会被创建。

复制线程

前面已经知道，复制需要 Master 和 Slave 有若干专门的线程。Master 的转储线程（dump thread）处理 Master 端的复制，而 Slave 的两个线程（I/O 线程和 SQL 线程）处理 Slave 端的复制。

- *Master* 转储线程
 当 Slave I/O 线程连接 Master 时，Master 将创建这个线程。转储线程负责从 Master 的 binlog 文件中读取记录，然后发送给 Slave。
 每个连接到 Master 的 Slave 对应一个转储线程。
- *Slave I/O* 线程
 这个线程负责连接 Master 并请求将所有改变转储（dump），然后写入中继日志，以便 SQL 线程进行进一步处理。
 每个 Slave 都有一个 I/O 线程。连接一旦建立就会一直保持，这样 Slave 就能立即收到 Master 的所有改变。
- *Slave SQL* 线程
 这个线程从中继日志中读取改变，然后在 Slave 数据库上应用这些改变。这个线程负责协调其他 MySQL 线程，保证这些改变不与 MySQL 服务器上的其他活动产生冲突。

从 Master 的角度来看，I/O 线程仅仅是一个客户端线程，能够执行 Master 的转储请求和 SQL 语句。也就是说客户端可以连接到服务器，伪装成 Slave，然后请求 Master 转储二进制日志中的改变。这就是 `mysqlbinlog` 程序（第 3 章中已经详细介绍过）的工作原理。

SQL 线程就是数据库的一个会话。也就是说，它像会话一样维护状态信息，但仍有一些区别。
201 因为 SQL 线程需要处理 Master 上多个线程的变化（Master 所有线程的事件都会按照提交的顺序写入二进制日志），所以 SQL 线程还保存了一些额外信息以正确区分这些事件。例如，由于临时表是会话特定的，所以要将不同会话的临时表分开，将 SessionID 添加到事件中去。然后 SQL 线程根据 SessionID 为 Master 的不同会话执行各自的任务。

关于 SQL 线程如何执行事件的细节将在本章后面介绍。



I/O 线程比 SQL 线程快得多，因为 I/O 线程仅将事件写入日志文件，而 SQL 线程必须知道如何执行数据库的各种改变。因此，在复制的过程中，有些事件通常会缓存在中继日志中。如果 Master 崩溃，那么在连接新的 Master 之前，还要处理崩溃问题。

为了避免这些缓存事件的丢失，要等待 SQL 线程处理完毕，Slave 才能尝试重新连接新的 Master。

后面将介绍几种方法来检查中继日志是否为空或者尚有事件需要执行。

Slave线程的启动和停止

第 2 章已经介绍了如何使用 `START SLAVE` 命令启动 Slave，但忽略了很多细节。现在我

们将深入描述 Slave 线程的启动和停止。

服务器启动时，如果存在 *master.info* 文件，还会同时启动 Slave 线程。前面提过，如果服务器建立了复制，即配置了服务器复制，并使用 STAT SLAVE 命令启动了 Slave 线程，那么 *master.info* 文件就会被创建。所以如果前面的会话是用于复制的，则从 *master.info* 和 *relay-log.info* 文件中存储的最后位置开始恢复复制，两个 Slave 线程的处理稍有不同：

- *Slave I/O* 线程
 Slave I/O 线程从 *master.info* 文件读取最后读位置进行恢复。
 写入事件时，I/O 线程将轮换（rotate）中继日志文件，即当到达指定文件容量后将开始写入一个新文件，然后更新位置信息。
- *Slave SQL* 线程
 Slave SQL 线程从 *relay-log.info* 文件读取中继日志位置进行恢复。

使用 STAT SLAVE 命令启动 Slave 线程，STOP SLAVE 命令停止 Slave 线程。使用下面的命令控制 Slave 线程，分别用来停止和启动 I/O 线程和 SQL 线程。

- START SLAVE 和 STOP SLAVE
 启动和停止 I/O 及 Slave 线程。
- START SLAVE IO_THREAD 和 STOP SLAVE IO_THREAD
 仅启动和停止 I/O 线程。
- START SLAVE SQL_THREAD 和 STOP SLAVE SQL_THREAD
 仅启动和停止 SQL 线程。

202

停止 Slave 线程时，复制的当前状态将保存到 *master.info* 和 *relay-log.info* 文件中。然后 Slave 线程再次启动时读取这些信息。



如果使用 master-host 参数（在 *my.cnf* 文件中设置，或者启动 mysqld 时作为参数传递）指定 Master 主机，则 Slave 也会启动。

不推荐直接使用 master-host 参数，而是在 CHANGE MASTER 命令中附加 MASTER_HOST 参数，所以这里就不介绍这个参数了。

通过Internet运行复制

很多原因导致需要在分隔两地的数据中心之间进行复制。其中一个原因是灾难恢复，例如地震或停电等。还可以定位到离用户较近的站点，例如内容交付网络（content delivery networks），从而提高响应时间。尽管拥有的资源足以让你租用专用光纤，我们

仍假定使用开放的 Internet 来连接。

Master 发给 Slave 的事件总是不安全的：事实上，很容易对其解码从中获得复制的信息。只要在防火墙内，并且不通过 Internet 复制（例如在两个数据中心之间进行复制），可能就够安全了。可是一旦你需要复制到另一个镇或大陆的数据中心，这时通过加密来保护信息不被窃取就变得尤为重要。

Internet 上的数据传输一般使用 SSL 进行加密。以下几种保护数据的方法都在某种程度上包含了 SSL：

- 使用服务器内置的加密支持，对 Master 到 Slave 的复制进行加密。
- 对于缺少 SSL 支持的程序，使用 Stunnel 程序建立一个 SSL 隧道（其实是一个虚拟的私有网络）。
- 在隧道模式下使用 ssh。

203 最后一种看上去并不比使用 Stunnel 更好，但如果机器上不能安装新程序且服务器上无法启用 ssh，这种方法就非常有用：使用 ssh 来建立一个隧道。这里我们不会深入讨论这种方法。

使用内置的 SSL 支持或者隧道来建立安全连接时，需要：

- 权威认证机构（CA）的证书；
- 服务器的（公有）证书；
- 服务器的（私有）密钥。

如何生成、管理和使用 SSL 证书的具体细节超出了本书的范畴。例 6-3 简单演示了如何生成自签名的公有证书及其相关的私有密钥。本例中假定使用 OpenSSL，其配置文件位于 */etc/ssl/openssl.cnf*。

例6-3：生成自签名的公有证书及其私有密钥

```
$ sudo openssl req -new -x509 -days 365 -nodes -config /etc/ssl/openssl.cnf\
>    -out /etc/ssl/certs/master.pem -keyout /etc/ssl/private/master.key
Generating a 1024 bit RSA private key
.....++++++
.++++++
writing new private key to '/etc/ssl/private/master.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
```

```
-----
Country Name (2 letter code) [AU]:SE
State or Province Name (full name) [Some-State]:Uppland
Locality Name (eg, city) []:Storvreta
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Big Inc.
Organizational Unit Name (eg, section) []:Database Management
Common Name (eg, YOUR name) []:master-1.example.com
Email Address []:mats@example.com
```

这个证书签名过程将自签名的公有证书放在 */etc/ssl/certs/master.pem*，私有密钥放在 */etc/ssl/private/master.key*（也用于为公有证书签名）。

同样，Slave 也需要创建服务器密钥和服务器证书。为了便于讨论，我们将 */etc/ssl/certs/slave.pem* 作为 Slave 服务器的公有证书名，*/etc/ssl/private/slave.key* 作为 Slave 服务器的私有密钥名。

使用内置支持建立安全复制

204

加密 Master 和 Slave 之间的连接，最简单的方法就是使用支持 SSL 的服务器。支持 SSL 的服务器编译方法超出了本书的范畴：如果感兴趣，请参照在线参考手册。

使用内置的 SSL 支持，需要做：

- 配置 Master，使 Master 的密钥可用。
- 配置 Slave，加密复制通道。

配置 Master 需要向 *my.cnf* 文件中添加以下参数：

```
[mysqld]
ssl-capath=/etc/ssl/certs
ssl-cert=/etc/ssl/certs/master.pem
ssl-key=/etc/ssl/private/master.key
```

`ssl-capath` 提供了可信 CA 证书所在的目录文件名，`ssl-cert` 给出了服务器证书的文件名，`ssl-key` 提供了服务器私有密钥的文件名。修改 *my.cnf* 文件以后必须重启服务器。

现在 Master 已经配置好了，可以为任何客户端提供 SSL 支持。由于 Slave 使用一般的客户端协议，所以还要保证 Slave 也能使用 SSL。

要配置 Slave 的 SSL 连接，发出 `CHANGE MASTER TO` 命令，附加 `MASTER_SSL` 参数，就可以开启 SSL 连接；然后使用 `MASTER_SSL_CAPATH`，`MASTER_SSL_CERT` 和 `MASTER_SSL_KEY` 命令，这些命令的功能与刚刚提到的配置参数 `ssl-capath`，`ssl-cert` 和 `ssl-key` 相同，但需要指定 Slave 端连接哪个 Master。

```
slave> CHANGE MASTER TO
    ->    MASTER_HOST = 'master-1',
    ->    MASTER_USER = 'repl_user',
    ->    MASTER_PASSWORD = 'xyzzy',
    ->    MASTER_SSL_CAPATH = '/etc/ssl/certs',
    ->    MASTER_SSL_CERT = '/etc/ssl/certs/slave.pem',
    ->    MASTER_SSL_KEY = '/etc/ssl/private/slave.key';
Query OK, 0 rows affected (0.00 sec)
slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

现在已经建立了 Slave 到 Master 之间的安全通道。

使用Stunnel建立安全复制

Stunnel 是一个简单易用的 SSL 隧道应用，可以在 SSL 服务器或 SSL 客户端上建立 Stunnel。

205 使用 Stunnel 建立安全连接与使用内置支持建立 SSL 连接差不多，只是需要更多的配置。如果服务器不支持 SSL 编译，或者由于某种原因你想将加密和解密数据所需的额外处理分离出去（只有在多核 CPU 的情况下才有意义），这种方法非常有用。

和内置支持一样，该方法也需要 CA 证书，而且每个服务器都要有公有证书和私有密钥。随后 Stunnel 命令需要使用这些证书和密钥，而服务器不用。

图 6-3 中描述了一个 Master，一个 Slave 和两个通过不安全网络通信的 Stunnel 实例。其中 Slave 服务器端的 Stunnel，通过标准的 MySQL 客户端连接，接收来自 Slave 服务器的数据，加密后发送给 Master 服务器端的 Stunnel 实例。Master 服务器端的 Stunnel 实例监听 SSL 端口，接收加密数据后将其解密，然后通过客户端连接将解密后的数据发送给 Master 服务器的非 SSL 端口。

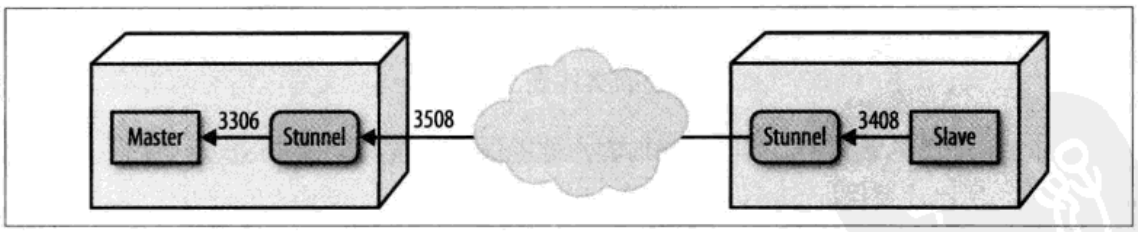


图6-3：通过不安全网络通信的Stunnel实例

建立 Stunnel 所需的配置文件如例 6-4 所示，其中 SSL 连接的监听套接字（socket）为 3508，Master 服务器监听的 MySQL 套接字默认为 3306。证书和密钥文件均使用前面例子中的文件名。

例6-4：Master服务器配置文件 */etc/stunnel/master.conf*

```
cert=/etc/ssl/certs/master.pem
key=/etc/ssl/private/master.key
CApath=/etc/ssl/certs
[mysqlrepl]
accept = 3508
connect = 3306
```

客户端建立 Stunnel 所需的配置文件如例 6-5 所示，其中 3408 端口为中间端口（Slave 连接本地 Stunnel 的非 SSL 端口），然后 Stunnel 连接 Master 服务器的 3508 端口，如上面例 6-4 所示。

例6-5：Slave服务器配置文件 */etc/stunnel/slave.conf*

```
cert=/etc/ssl/certs/slave.pem
key=/etc/ssl/private/slave.key
CApath=/etc/ssl/certs
[mysqlrepl]
accept = 3408
connect = master-1:3508
```


现在每台服务器都可以启动 Stunnel 程序了，然后配置 Slave 连接 Slave 服务器端的 Stunnel 实例。由于这个 Stunnel 实例运行在与 Slave 相同的服务器上，所以将 localhost 设置为要连接的 Master 主机，其连接的端口号为 3408。然后由 Stunnel 管理与 Master 服务器的连接隧道。

```
slave> CHANGE MASTER TO
    ->    MASTER_HOST = 'localhost',
    ->    MASTER_PORT = 3408,
    ->    MASTER_USER = 'repl_user',
    ->    MASTER_PASSWORD = 'xyzzy';
Query OK, 0 rows affected (0.00 sec)

slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

这样你就完成了在不安全网络中建立安全连接。



如果使用基于 Debian 的 Linux（如 Debian 或 Ubuntu），将 */etc/default/stunnel4* 文件设置为 ENABLED=1，可以为每个配置文件启动一个 Stunnel 实例（配置文件位于 */etc/stunnel* 目录）。

所以，如果按照本节介绍的方法创建 Stunnel 配置文件，无论什么时候启动机器，都会自动启动一个 Slave Stunnel 实例和一个 Master Stunnel 实例。

细粒度控制复制

理解了复制的内部原理和复制所用的信息，就可以更好地控制复制，学会如何避免可能遇到的问题。本节将介绍一些有用的背景知识。

关于复制状态的信息

关于复制状态的信息大部分都在 Slave 上，但 Master 上也有一些。Master 上的信息大多与 binlog（第 3 章中有介绍）相关，当然还有已连接的 Slave 的相关信息。

`SHOW SLAVE HOSTS` 命令仅显示使用 `report-host` 参数的 Slave 信息，Slave 使用该参数将信息提供给所连接的 Master。Master 不能相信已连接的 Slave 提供的信息，因为
207 Master 和 Slave 之间使用的是 NAT 路由器。除了主机名以外，Slave 提供的信息还有：

- `report-host`
 已连接的 Slave 主机名，通常是 Slave 的域名或其他类似的标识符，实际上可以是任意字符串。例 6-6 中我们使用的是“Magic Slave”。
- `report-port`
 Slave 用于监听连接的端口，默认是 3306。
- `report-user`
 连接 Master 的用户。该值不需要与 `CHANGE MASTER TO` 的值相匹配。只有服务器使用了 `show-slave-auth-info` 参数时才有这个参数。
- `report-password`
 连接 Master 时所使用的密码。该值不需要与 `CHANGE MASTER TO` 的密码相匹配。
- `show-slave-auth-info`
 如果启用了这个参数，`SHOW SLAVE HOSTS` 命令将输出更多用户和密码的相关信息。

例 6-6 给出了 `SHOW SLAVE HOSTS` 命令的输出示例，其中 Master 连接了三个 Slave。

例6-6：SHOW SLAVE HOSTS命令的输出示例

```
master> SHOW SLAVE HOSTS;
+-----------+-------------+------+------------------+-----------+
| Server_id | Host        | Port | Rpl_recovery_rank | Master_id |
+-----------+-------------+------+------------------+-----------+
|         2 | slave-1     | 3306 |                0 |         1 |
|         3 | slave-2     | 3306 |                0 |         1 |
|         4 | Magic Slave | 3306 |                0 |         1 |
+-----------+-------------+------+------------------+-----------+
1 row in set (0.00 sec)
```

这个命令输出连接到 Master 的 Slave 信息，包括通过中继间接连接到 Master 的 Slave。

如果启用 show-slave-auth-info 参数，还会输出两个字段（这里我们没有列出）。

这些字段仅仅提供信息，不一定表示真实的 Slave 主机名或端口，也不一定是使用 CHANGE MASTER TO 配置 Slave 时所需的用户和密码。

- Server_id
 已连接的 Slave 的服务器 ID。
- Host
 report-host 参数指定的主机名。
- User
 report-user 参数指定的用户名。
- Password
 report-password 参数指定的密码。
- Port
 端口号。
- Master_id
 Slave 从哪个服务器进行复制的，即 Master ID。
- Rpl_recovery_rank
 从未用过，MySQL 5.5 版本已经去掉了这个字段。



间接连接的 Slave 的信息并不完全可信，因为增加 Slave 可能使信息变得不准确。所以，需要去掉间接连接的 Slave 信息，只显示直接连接的 Slave，因为它们的信息是可信的。

使用 SHOW MASTER LOGS 命令查看 Master 跟踪了哪些二进制日志文件。该命令的典型输出参见例 6-7。

例6-7：SHOW MASTER LOGS命令的输出

```
master> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| master-bin.000011 |    469768 |
| master-bin.000012 |   1254768 |
| master-bin.000013 |    474768 |
| master-bin.000014 |      4768 |
+------------------+-----------+
1 row in set (0.00 sec)
```

SHOW MASTER STATUS 命令（例 6-8）输出下一个事件即将写入二进制日志的位置。由于

master 只有一个 binlog 文件，所以该表总是只有一行记录。因此，SHOW MASTER LOGS 输出的最后一行正好与该命令的输出匹配，只是显示的字段不同。也就是说，如果需要使用 SHOW MASTER LOGS 命令来实现某些特性，则不必同时执行 SHOW MASTER STATUS，只要参照 SHOW MASTER LOGS 的最后一行即可。

209 例6-8：SHOW MASTER STATUS命令的典型输出

```
master> SHOW MASTER STATUS;
+-------------------+----------+--------------+------------------+
| File              | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-------------------+----------+--------------+------------------+
| master-bin.000014 |     4768 |              |                  |
+-------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

SHOW SLAVE STATUS 命令用于查看 Slave 线程的状态。该命令几乎包含了复制状态的所有信息。下面详细地看看这个命令的输出，如例 6-9 所示。

例6-9：SHOW SLAVE STATUS命令的典型输出

```
               Slave_IO_State: Waiting for master to send event
                  Master_Host: master1.example.com
                  Master_User: repl_user
                  Master_Port: 3306
                Connect_Retry: 1
              Master_Log_File: master-bin.000001
          Read_Master_Log_Pos: 192
               Relay_Log_File: slave-relay-bin.000006
                Relay_Log_Pos: 252
        Relay_Master_Log_File: master-bin.000001
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 192
              Relay_Log_Space: 553
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
```

```
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
```


I/O和SQL线程的状态

Slave_IO_Running 和 Slave_SQL_Running 两个字段分别表示 Slave I/O 线程或 SQL 线程是否正在运行。如果 Slave 线程没有运行，可能是线程已经停止，或者复制过程出错。

如果 I/O 线程没有运行，Last_IO_Errno 和 Last_IO_Error 字段将给出线程停止的原因。同样，Last_SQL_Errno 和 Last_SQL_Error 字段将给出 SQL 线程停止的原因。如果线程在没有出错的情况下停止，比如，显式停止线程，或者达到了循环终止的条件，这时没有出错信息，error 字段为 0。这时输出结果类似于例 6-9，其中 Last_Errno 和 Last_Error 字段分别对应于 Last_SQL_Errno 和 Last_SQL_Error。

Slave_IO_State 描述了当前运行的 I/O 线程的状态。随着 I/O 线程的状态不同，其消息变化的状态图如图 6-4 所示。

这些消息的含义如下：

- 等待 Master 更新
 I/O 线程初始化以后，试图建立 Master 连接之前，将短暂出现这个消息。
- 连接 Master
 Slave 正在尝试连接 Master，但连接尚未建立时，出现这个消息。
- 检查 Master 的版本
 Slave 已经连接上 Master，正在与 Master 进行握手（handshake）时，出现这个消息。
- 在 Master 上注册 Slave
 Slave 试图在 Master 上注册时出现这个消息。注册时，Slave 将 report-host 的值发送给 Master。通常是 Slave 的主机名或 IP 地址，但也可以是任意字符串。Master 不能仅仅依靠检查 TCP 连接的 IP 地址，因为 Master 和 Slave 之间可能有 NAT（网络地址转换）路由器。
- 请求 binlog 转储
 当 Slave 开始向 Master 请求 binlog 转储时出现这个消息。Slave 将 binlog 文件、binlog 位置和服务器 ID 发送给 Master。

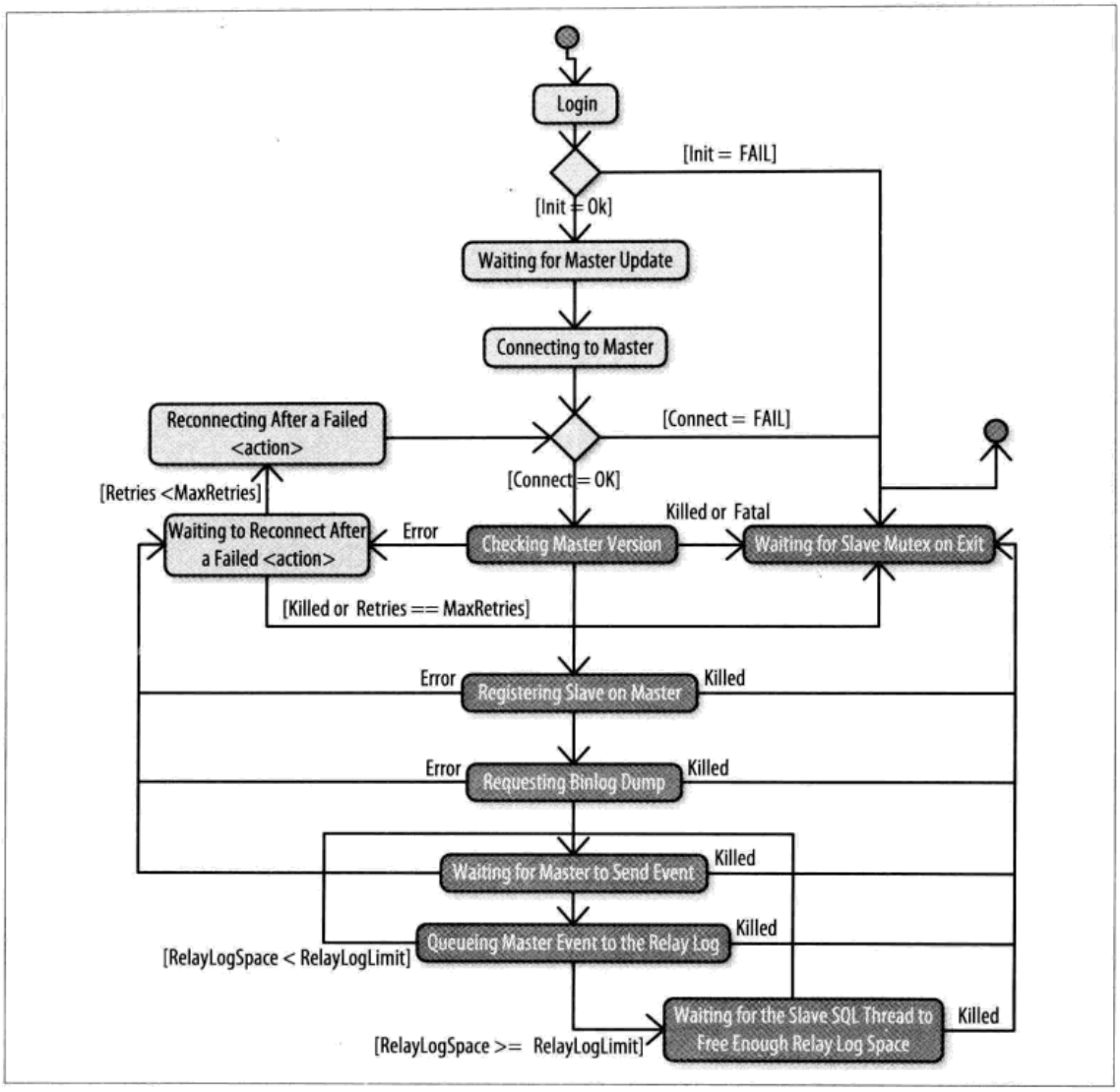


图6-4：Slave I/O线程的状态

- 等待 Master 发送事件

 Slave 已经连接上 Master，并等待 Master 发送事件时出现这个消息。

211

- 以队列的形式将 Master 事件写入中继日志

 Slave I/O 线程正要将 Master 发送的事件写入中继日志时出现这个消息。不管事件真的被写入中继日志还是被跳过，都会出现这个消息，其规则参见前面的“过滤复制事件”一节。

212



使用脚本或其他工具查看这些消息时，弄清楚消息中蕴含的真实内容非常重要，而不仅仅局限于其字面意思。

- 操作后等待重新连接
 前面的操作出错失败后，Slave 尝试重新连接时出现这个消息。这些操作可能为：
 在 Master 上注册
 当 Slave 试图在 Master 上注册时。
 Binlog 转储请求
 当 Slave 从 Master 请求 binlog 转储时。
 读取 Master 事件
 当 Slave 等待或读取来自 Master 的事件时。
- 操作失败后重新连接
 Slave 操作失败后尝试重新连接 Master，但尚未建立连接时出现这个消息。这些操作的可能值与“操作后等待重新连接”消息的值一样。
- Slave 等待退出
 关闭 I/O 线程时出现该消息。
- 等待 Slave SQL 线程释放中继日志空间
 中继日志的空间大小达到限定值（由 relay-log-space-limit 参数设置）时出现这个消息，SQL 线程需要处理某些中继日志以便写入新事件。

二进制日志和中继日志的位置

Slave 上的复制在处理事件时，需要维护三个位置。

使用 SHOW SLAVE STATUS 命令查看这些位置，如例 6-9 所示，其中包含以下几对字段。

- Master_Log_File，Read_Master_Log_Pos
 Master 的读取位置：I/O 线程即将从 Master 二进制日志读取的下一个事件的位置。
 这些字段值来自 master.info 文件的第 2、3 行，如例 6-1 所示。
- Relay_Master_Log_File，Exec_Master_Log_Pos
 Master 的执行位置：SQL 线程即将执行的 master binlog 中下一个事件的位置。
 这些字段值来自 *relay-log.info* 文件的第 3 行，如例 6-2 所示。
- Relay_Log_File，Relay_Log_Pos
 中继日志的执行位置：SQL 线程即将执行的 Slave 中继日志中下一个事件的位置。
 这些字段值来自 *relay-log.info* 文件的第 1、2 行，如例 6-2 所示。

213

通过这些位置可以获得复制过程的相关信息，或用来优化第 4 章中的某些算法。

例如，通过比较 Master 的读取位置和 Master 的执行位置，可以确定是否有事件等待执行。如果 I/O 线程已经停止，只要等待中继日志为空就行了：一旦位置相同，中继日志中就没有等待的事件，从而 Slave 可以安全地停止并重定向到新 Master。

例 6-10 给出了等待 Slave 中继日志为空的示例代码。MySQL 提供了一个方便的函数 MASTER_POS_WAIT，负责等待 Slave 中继日志处理完所有等待中的事件。

例6-10：等待中继日志变为空的Python脚本

```
class SlaveNotRunning(Error):
    "Exception raised when slave is not running but were expected to run"
    pass
def slave_wait_for_empty_relay_log(slave):
    result = server.sql("SHOW SLAVE STATUS");
    file = result["Master_Log_File"]
    pos = result["Read_Master_Log_Pos"]
    if server.sql(_MASTER_POS_WAIT, (file, pos)) is None:
        raise SlaveNotRunning
```

使用这些位置还可以优化第 4 章中描述的场景。例 4-15 给出了将 Slave 提升为 Master 的例子，将某个 Slave 切换为新 Master 前，可能还要处理其他 Slave 中继日志的事件。而且，保证已提升的 Slave 在允许其他 Slave 连接前已经执行了所有事件，可以将数据丢失减到最小。

修改例 4-15 得到例 6-11，切换前，保证原来的 Slave 将其中继日志中的所有事件执行完毕。

214

例6-11：提升Slave时将事件丢失减小到最小

```
def fetch_global_trans_id(slave):
    result = slave.sql("SELECT server_id, trans_id FROM Last_Exec_Trans")
    return (int(result["server_id"]), int(result["trans_id"]))

def wait_for_empty_relay_log(slave):
     result = slave.sql("SHOW SLAVE STATUS")
     slave.sql("SELECT MASTER_POS_WAIT(%s,%s)",
               (result["Master_Log_File"], result["Read_Master_Log_Pos"]))

def promote_slave(slaves):
    slave_info = {}

    # Collect the global transaction ID of each slave
    for slave in slaves:
        slave.connect()
        wait_for_empty_relay_log(slave)
        server_id, trans_id = fetch_global_trans_id(slave)
        slave_info.setdefault(trans_id, []).append((server_id, trans_id, slave))
        slave.disconnect()

    # Pick the slave to promote by taking the slave with the highest
    # global transaction id.
    new_master = slave_info[max(slave_info)].pop()[2]

    def maybe_change_master(server_id, trans_id, position):
```

```
        from mysqlrep.utility import change_master
        try:
            for sid, tid, slave in slave_info[trans_id]:
                if slave is not new_master:
                    change_master(slave, new_master, position)
        except KeyError:
            pass

    # Read the the master log files of the new master.
    new_master.connect()
    logs = [row["Log_name"] for row in new_master.sql("SHOW MASTER LOGS")]
    new_master.disconnect()

    # Read the master log files one by one in reverse order, the
    # latest binlog file first.
    logs.reverse()
    for log in logs:
        scan_logfile(new_master, log, maybe_change_master)
```

除了这里所介绍的技术外，有些书中还提到了另一种技术，即使用 SHOW PROCESSLIST 命令检查 SQL 线程的状态。如果“State”字段为“Has read all relay log; waiting for the slave I/O thread to update it”，那么 SQL 线程已经读取了整个中继日志。这个 State 消息只能由 SQL 线程产生，所以你可以放心地在所有线程中查找这个消息。

处理断开连接的参数

由图 6-4 可知，I/O 线程负责维护 Slave 与 Master 之间的连接，其中包含很多复杂的逻辑。如果 I/O 线程丢失了 Master 连接，则进行有限次尝试重新连接 Master。如果某个时间段内没响应，则 I/O 线程重新尝试下一次连接。重试的时间间隔和重试次数由三个参数控制：

215

- `--slave-net-timeout`
 可接受的无响应时间，超过这个时间 Slave 认为 Master 连接已经丢失并尝试重新连接。如果明显知道连接已经断开，则该参数没有用。这时，Slave I/O 线程立即尝试重新连接（等待的时间取决于 `master-connect-retry` 的值，且尝试次数不得超过 `master-retry-count`）。
 默认值为 3600 秒。
- `--master-connect-retry`
 两次尝试间隔的秒数。可以将这个参数设为 CHANGE MASTER TO 命令的 CONNECT_RETRY 参数值。*my.cnf* 文件中不要使用这个参数。
 默认值为 60 秒。
- `--master-retry-count`
 最大尝试次数。

默认值为 86400。

默认值可能不是最佳的，所以最好提供自定义的值。

Slave是如何处理事件的

复制的核心是日志事件：它是复制系统的信息携带者，包含所有必需的元数据，保证复制可以执行 Master 的所有变化从而生成 Master 的副本。因为 Master 二进制日志中的事务是按照提交顺序执行的，所以各个事务在 Slave 上以同样的顺序执行，得到相同的结果。

Slave SQL 线程顺序执行 Master 的所有会话事件。Slave 执行事件的方式不同将产生不同的结果。

- Slave 是单线程的，而 Master 是多线程的
 日志事件在 Slave 上以单线程执行，但在 Master 上是多线程。这样，如果 Master 上提交了很多事务，Slave 就难以与 Master 保持同步了。
216
- 有些语句是特定于会话的
 Master 上的有些语句是特定于会话的，这样 Slave 的单线程执行可能产生不同的结果：
 - 每个用户变量都是特定于会话的。
 - 临时表是特定于会话的。
 - 有些函数也是特定于会话的——例如 `CONNECTION_ID`。
- 二进制日志决定了执行顺序
 尽管二进制日志的事务看上去相互独立（理论上可以并行执行），但实际上并不是这样。也就是说，Slave 必须按顺序执行事务，才能保证 Master 和 Slave 一致。

管理I/O线程

尽管大部分的事件处理由 SQL 线程完成，但是在事件到达 SQL 线程之前，还需要 I/O 线程做一些管理工作。所以在讨论 SQL 线程“真正执行”之前，先来看看 I/O 线程如何处理。为了获得较高的处理速度，I/O 线程只使用某些字节来判断事件的类型，必要时将事件写入中继日志：

- 停止事件
 该事件表明 Slave 链中的下一个服务器被有序地停止。I/O 线程不把这个事件写入中继日志。
- 轮换（Rotate）事件
 如果 Master 的二进制日志被轮换，中继日志也要轮换。中继日志轮换的次数可能比 Master 多，但是每次 Master 的二进制日志轮换时，中继日志至少轮换一次。

- 格式化描述事件
 中继日志轮换时保存该事件。回想一下，两个连续的 binlog 文件的格式可能不同，所以 I/O 线程需要保存该事件以正确地处理文件。

如果是环形复制，或者双主配置（dual-master setup）（即只在两个服务器之间进行环形复制），事件将一直在环中传送下去直到到达最初的目标服务器为止。为了避免事件在环中无休止地复制，必须将以前执行过的事件删除。

为此，每个服务器需要检查事件是否包含该服务器本身的 ID。如果有，说明这个事件原来就是这个服务器发送的，已经在整个环上复制了一圈。为了避免事件无限复制（因此也会无限应用），不把这个事件写入中继日志，而是忽略它。这可以通过服务器设置 `replicate-same-server-id` 参数实现。如果设置了该参数，服务器将不再检查服务器 ID，这样不管服务器 ID 是什么，事件都会写入中继日志。 217

SQL线程的处理

Slave SQL 线程读取中继日志，然后在 Slave 上重新执行 Master 的数据库语句。有些事件还需要 SQL 语句以外的特殊信息，包括：

- 将 *Master* 的上下文发送给 *Slave* 服务器
 有时候 Slave 需要状态信息才能正确执行语句。第 3 章中提到过，Master 通过写一个或多个上下文事件来传递额外信息。有些信息是属于特定线程的，但不同于下面一项中的线程信息。
- 处理不同线程的事件
 由于 Master 执行的事务来自若干个会话，Slave SQL 线程必须知道事件是由哪个线程产生的。因为 Master 最了解语句的情况，所以由 Master 将事件标记为某个线程的。例如，通常 Master 将操作临时表的事件标记为某个线程的。
- 过滤事件和表
 SQL 线程负责 Slave 上的过滤处理。MySQL 提供了数据库过滤（通过 `replicate-do-db` 和 `replicate-ignore-db` 设置）和表过滤（通过 `replicate-do-table`, `replicate-ignore-table`, `replicate-wild-do-table` 和 `replicate-wild-ignore-table` 设置）。
- 跳过事件
 在处理复制停止后的恢复时，可以在重启复制时选择跳过事件。SQL 线程处理事件的跳过。

上下文事件

Master 上的有些事件需要上下文才能正确执行。上下文通常是指线程特定的特性，比如用户自定义的变量，也包括正确执行所需的状态信息，比如带有自动增量字段的表的自

动增量值。要将上下文发送给 Slave，Master 需要向二进制日志中写入一组上下文事件。

Master 在写事件之前先将上下文事件写入二进制日志。目前，上下文事件只与 Query 事件相关，它在 Query 事件之前写入二进制日志。

218 上下文事件可以分为以下几类：

- 用户变量事件

 该事件包含用户自定义的变量名及变量值。

 不论语句中有没有用户自定义变量的引用，都会产生这个事件。

```
SET @foo = 'SmoothNoodleMaps';
INSERT INTO my_albums(artist, album) VALUES ('Devo', @foo);
```

- 整型变量事件

 该事件包含 INSERT_ID 或 LAST_INSERT_ID 会话变量的整型值。

 INSERT_ID 整型变量事件用于含 AUTO_INCREMENT 列的表的插入语句，它传送 AUTO_INCREMENT 列的下一个值。例如，下面这个表的定义语句需要该信息：

```
CREATE TABLE Artist (id INT AUTO_INCREMENT PRIMARY KEY, artist TEXT);
INSERT INTO Artist VALUES (DEFAULT, 'The The');
```

 当语句使用 LAST_INSERT_ID() 函数时，产生 LAST_INSERT_ID 整型变量事件，例如：

```
INSERT INTO Album VALUES (LAST_INSERT_ID(), 'Mind Bomb');
```

- 随机事件

 如果语句中调用了 RAND() 函数，则事件中含有随机种子，允许 Slave 重新产生 Master 上生成的“随机”值。

```
INSERT INTO my_table VALUES (RAND());
```

前面描述的情况需要上下文事件来保证正确执行，但有时候无法使用上下文事件解决。例如，复制系统不能处理用户自定义的函数（user-defined function，UDF），除非 UDF 是确定性的，并且 Slave 上也有这个函数，这时可以用用户变量事件解决。

用户变量事件还用于：避免非确定性函数的复制问题、提高性能，以及完整性检查等。

例如，假定要将文档存入数据表。使用 AUTO_INCREMENT 为每个文档自动分配一个序号。为了维护文档的完整性，还要添加文档的 MD5 校验和。表定义语句如例 6-12 所示。

219 例6-12：带有MD5校验和的文档表

```
CREATE TABLE document(
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    doc BLOB,
    checksum CHAR(32)
);
```

将文档及其校验和存储在同一张表中，验证了文档的完整性，保证文档不被损坏，如例

6-13 所示。尽管目前 MD5 校验和并不绝对安全，但还是避免了某些随机错误，比如硬盘和内存问题等。

例6-13：向表中插入记录，并检查文档的完整性

```
master> INSERT INTO document(doc) VALUES (document);
Query OK, 1 row affected (0.02 sec)
master> UPDATE document SET checksum = MD5(doc) WHERE id = LAST_INSERT_ID();
Query OK, 1 row affected (0.04 sec)
master> SELECT id,
    ->        IF(MD5(doc) = checksum, 'OK', 'CORRUPT!') AS Status
    ->   FROM document;
+-----+----------+
|  id |   Status |
+-----+----------+
|   1 | OK       |
|   2 | OK       |
|   3 | OK       |
|   4 | OK       |
|   5 | OK       |
|   6 | OK       |
|   7 | CORRUPT! |
|   8 | OK       |
|   9 | OK       |
|  10 | OK       |
|  11 | OK       |
+-----+----------+
11 row in set (5.75 sec)
```

但是怎么用于复制呢？这取决于你怎么使用它。例 6-13 中执行 INSERT 语句时，该语句会被写入二进制日志，也就是说 Slave 将重新计算 MD5 校验和。那么，如果文档在发给 Slave 时损坏怎么办？将会对已损坏的文档重新计算 MD5 校验和，这样就不会发现文档已经损坏。所以例 6-13 中的语句对于复制来说是不安全的。不过，我们可以改进。

按照例 6-14 那样，将校验和存在一个用户自定义的变量中，然后在 INSERT 语句中使用这个变量。由于用户自定义的变量存的是 MD5 函数计算的真实值，所以即使文档在传输过程中损坏（当然校验和并没有在传输时损坏），Master 和 Slave 也是完全一样的。无论哪种方法，总会发现复制时的文档损坏。

220

例6-14：向表中插入文档的复制安全（Replication-safe）方法

```
master> INSERT INTO document(doc) VALUES (document);
Query OK, 1 row affected (0.02 sec)
master> SELECT MD5(doc) INTO @checksum FROM document WHERE id = LAST_INSERT_ID();
Query OK, 0 rows affected (0.00 sec)
master> UPDATE document SET checksum = @checksum WHERE id = LAST_INSERT_ID();
Query OK, 1 row affected (0.04 sec)
```

线程特定事件

前面提过，有些语句是特定于线程的，在不同线程中执行时可能产生不同的结果。原因是：

- 读写线程本地对象
 线程本地对象（thread-local object）可能与另一个线程中的对象产生命名冲突。这种对象的典型例子是临时表或用户自定义变量。
 我们已经知道复制如何处理用户自定义变量，所以本节仅介绍临时表的处理。
- 使用具有线程特定结果的变量或函数
 有些变量和函数由于运行的线程不同而导致值不同。典型的例子是服务器变量 `connect_id`。

这两种情况下，服务器的处理大不相同。另外，有时候复制无法区分服务器和客户端，所以结果可能稍有不同。

处理线程本地对象需要线程本地存储（thread-local store，TLS），但是，由于 Slave 是单线程执行，所以还需要管理存储，并保持 TLS 的独立。Slave 根据服务器进程 ID，线程 ID 和线程的序列号，为临时表创建一个唯一的文件名。也就是说例 6-15 中的两个语句（由 Master 上的不同客户端运行）将在 Slave 上创建两个不同的文件名来代表临时表。

例6-15：两个线程，每个线程创建一个临时表

```
master-1> CREATE TEMPORARY TABLE cache (a INT, b INT);
Query OK, 0 rows affected (0.01 sec)
master-2> CREATE TEMPORARY TABLE cache (a INT, b INT);
Query OK, 0 rows affected (0.01 sec)
```

221 由于 Master 所有线程的所有语句都是按顺序存储在二进制日志中的，所以需要区分这两个语句。否则，Slave 执行时将产生错误。

为了区分二进制日志中的语句以防止冲突，服务器将包含该语句的 Query 事件标记为某个线程的，并将线程 ID 添加到该事件。实际上，线程 ID 是加到所有的 Query 事件的，但只有线程特定的语句才需要这么做。

Slave 接收到一个特定线程的事件时，将设置一个特定于复制 Slave 线程的变量，即 *pseudothread ID*，它与事件的线程 ID 有关。然后使用 pseudothread ID 创建临时表，使用 Slave 服务器的进程 ID 创建文件名——对所有的 Master 线程来说都一样，只要不同线程的表不同就没关系了。

我们还说过特定线程的函数和变量在复制时需要特别对待，但它们并不是由服务器处理的。如果语句引用了服务器变量，服务器变量的值将在 Slave 上读取。如果你想复制完

全相同的值，必须将该值存储在用户自定义变量中，如例 6-14 所示，也可以使用基于行的复制，这将在后面讲到。

过滤和跳过事件

某些情况下，使用复制过滤器或 Slave 已明确指示要跳过一些事件，所以事件可能被跳过。

SQL_SLAVE_SKIP_COUNTER 变量指定 Slave 服务器跳过的事件的数目。不要在 SQL 线程正在运行的时候设置该变量。这个条件很容易满足，因为通常使用这个变量来跳过那些已经使复制停止的事件。

当然，需要审查和处理导致复制停止的错误，但是如果手动解决这些问题，必须忽略导致停止复制的事件，然后在事件结束后强制复制继续。这个变量很方便，不需要使用 CHANGE MASTER TO 命令。例 6-16 给出了在一个错误语句导致复制停止后使用该功能的例子。

例6-16：使用SQL_SLAVE_SKIP_COUNTER

```
slave> SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 3;
Query OK, 0 rows affected (0.02 sec)

slave> START SLAVE;
Query OK, 0 rows affected (0.02 sec)
```

在启动 Slave 时，恢复复制之前需要跳过三个事件。如果跳过这三个事件会导致某个事务中断，那么 Slave 事务执行结束后再继续跳过事件。 222

如果建立了复制过滤器，还可以由 Slave 进行事件过滤。第 3 章中讨论过，Master 可以处理过滤，但如果有 Slave 过滤器，事件将在 SQL 线程中过滤，即事件仍来自 Master 并存储在中继日志中。

数据库过滤器和表过滤器的过滤过程不同。对特定数据库来说，判断语句是否应该从二进制日志中过滤的逻辑在第 3 章中已经详细介绍过。这个逻辑同样适用于 Slave 过滤器，不过还需要处理一组表过滤器。

重要的是，应用在单个表上的过滤器将导致与该过滤器有关的整个语句不再参与复制。Slave 过滤语句的逻辑如图 6-5 所示。

表的过滤很容易变得难以理解，为了避免得到不想要的结果，建议使用以下规则：

- 不要限定数据表所在的数据库。在语句前使用 USE 语句，设定一个新的默认数据库。
- 不要在单个语句中更新不同数据库中的表。

- 不要在单个语句中更新多个表，除非你知道所有的表都（不）被过滤。注意图 6-5 中的逻辑是，即使只有一个表被过滤，整个语句也将被过滤。

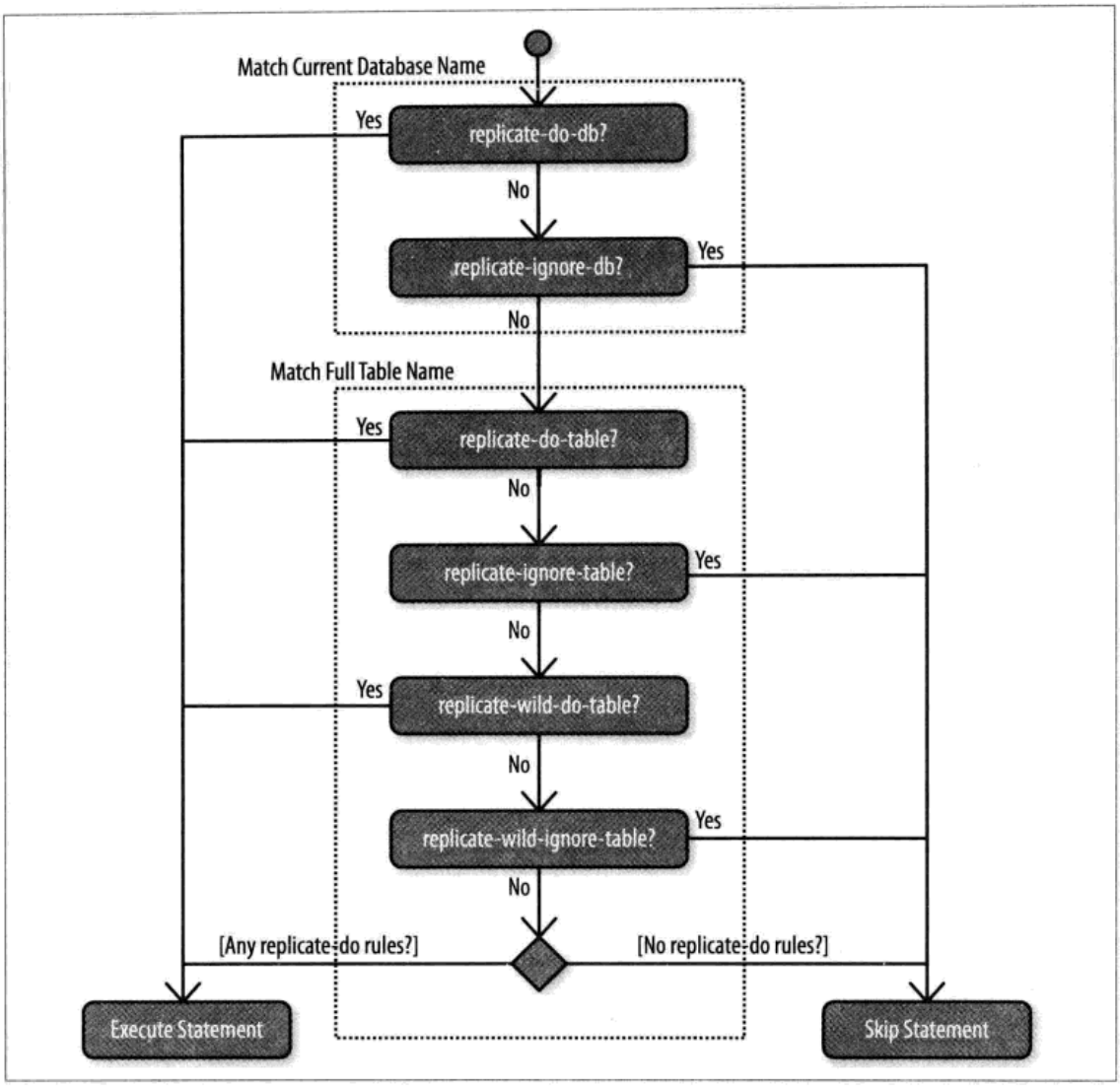


图6-5：复制过滤规则

Slave的安全和恢复

Slave 服务器也有可能崩溃，如果服务器崩溃，必须进行恢复。如果 Slave 崩溃，首先要搞清楚服务器崩溃的原因。这个过程无法自动完成，因为崩溃的原因很多且无法预测。可能是磁盘容量不足，读取损坏事件，或者由于某种原因重新执行语句，导致重复关键字错误等。但是，某些恢复过程可以自动执行，而且可以自动诊断错误。

同步，事务和数据库崩溃问题

为了保证 Master 或 Slave 崩溃以后 Slave 能够安全地恢复复制，需要考虑以下两个问题：

- 保证 Slave 上存有恢复所需的所有数据；
- 执行 Slave 的恢复。

Slave 通过磁盘同步尽量满足第一个条件。操作系统将所需的文件存在内存中，并周期性地（或强制）写入磁盘，从而提供了可接受的性能。也就是说，写入文件的数据并不安全，一旦发生崩溃，内存中的数据将丢失。 223

为了强制 Slave 将文件写入磁盘，数据库服务器发出 `fsync` 调用，将所有内存中的数据写入磁盘。为了保护复制数据，通常 MySQL 服务器定期在中继日志、*master.info* 文件和 *relay-log.info* 文件上执行 `fsync` 调用。

I/O线程同步

对于 I/O 线程来说，无论什么时候处理事件都有两个 `fsync` 调用：一个将中继日志刷新(flush)到磁盘，另一个将 *master.info* 文件刷新到磁盘。这种刷新顺序保证了没有事件丢失，即使 Slave 在刷新中继日志和刷新 *master.info* 文件之间的某个时候崩溃。但是，如果发生以下情况，可能产生重复事件：

- 服务器刷新中继日志，正要更新 *master.info* 文件中的 Master 读位置时。
- 服务器崩溃，也就是说 Master 读位置现在指向事件被刷新到中继日志之前的位置。
- 服务器重启，并从 *master.info* 获得 Master 读位置，即在最后一个事件写入中继日志之前的位置。
- 从这个位置恢复复制，导致事件重复。

如果按相反顺序刷新文件（先 *master.info* 文件再中继日志），可能丢失事件，因为 Slave 向中继日志写事件之后将执行复制的恢复。我们认为丢失事件比重复事件严重，因此要先刷新中继日志。

SQL线程同步

224

SQL 线程轮流处理每个事件来处理中继日志中的组。在处理组中的所有事件时，SQL 线程按以下方式提交事务：

1. 将事务提交到存储引擎（假定存储引擎支持提交）。
2. 更新 *relay-log.info* 文件的下一个事件的位置，这个位置也是处理下一个组的开始位置。

3. 发出 fsync(2) 调用，将 *relay-log.info* 文件写入磁盘。

在组内执行时，通过增加事件的位置，线程跟踪 SQL 线程对中继日志的读取位置。但是如果发生崩溃，将从 *relay-log.info* 文件的最后一个记录位置恢复执行。

这种方式使得 SQL 线程的原子更新问题与前面的 I/O 线程不同，所以下面的几种情况可能导致 Slave 数据库和 *relay-log.info* 文件不同步：

1. 事件在数据库上已应用，且事务已提交。下一步是更新 *relay-log.info* 文件。
2. Slave 崩溃，也就是说，*relay-log.info* 文件现在指向刚刚完成的事务的开始。

225

3. 恢复时，SQL 线程从 *relay-log.info* 文件读取信息，并从保存的位置开始复制。
4. 重复上一次执行的事务。

所有这些归结为在 Slave 上提交事务和更新复制信息并不是原子性操作：*relay-log.info* 文件并不能准确地反映数据库上提交的事务。

总之，复制系统认为表（甚至非事务表）上的每个语句都会自动执行，从这个意义上说，它是崩溃安全的（crash-safe）。这意味着要么整条语句都被执行，要么完全不执行。所以，如果在语句执行过程中发生崩溃，存储引擎必须保证服务器重启时将其回滚。

保护非事务性语句的规则

崩溃后重新执行时不会跟踪和保护事务之外的语句。Master 和 Slave 上的问题类似。如果 MyISAM 表上的语句由于 Master 崩溃而中断，语句将不再记入日志，因为语句是在执行完成之后才写日志的。重启（修复成功）时 MyISAM 表将部分更新，但是二进制日志将不再记录该语句。

Slave 上的情形类似：如果语句（或更改非事务性表的事务）在执行过程中发生崩溃，表可能变化，但是组的位置尚未发生改变。Slave 再次重启复制时将重新执行这个非事务性语句。

在更新非事务性表的过程中，导致崩溃的原因很难被自动发现。但是通过观察一些规则，可以保证出现问题时至少接收到一个错误信息。

- INSERT 语句
 要复制的表中必须有主键。这样，重新执行的 INSERT 语句将会产生一个重复键错误，然后停止 Slave，你可以检查 Master 和 Slave 不一致的原因。
- DELETE 语句
 避免使用 LIMIT 从句。如果不用 LIMIT 从句，语句将删除相同的行（即匹配 WHERE

从句的那些行）或者删除尚未删除的行，或者什么都不做，因为所有行已经被删除。但是，如果语句中有 LIMIT 从句，则只删除符合 WHERE 条件的行子集。如果再次执行同样的语句，将删除另一个行子集。

- UPDATE 语句

 这种语句最复杂。语句必须是幂等的（两次执行得到同样的结果）或者语句两次执行结果的偶然性是可接受的，比如用于页面访问维护统计数据的 UPDATE 语句。

226

多源复制

你可能注意到，一个 Slave 连接多个 Master 并接收来自所有 Master 的变化，是不可能的。这种拓扑称为多源（*multisource*），不要与第 5 章中的多主（multimaster）拓扑混淆。多源拓扑中，变化来自多个 Master；而多主拓扑中，通过将每个 Master 上的变化复制到所有其他 Master，使得服务器组扮演单个 Master 的角色。

将多源复制引入 MySQL 计划已久，但设计上有个问题：如何处理更新冲突。不同的源同时发生改变，或者两个中继服务器同时传送某个 Master 的变化，都会产生冲突。图 6-6 解释了这两种冲突。第一种，两个 Master（源）同时改变同样的数据，Slave 无法判断数据最终如何变化。第二种，只有一个 Master 改变，但 Slave 接收到两个改变。这两种情况下，Slave 都无法区分接收到的两个中继事件，所以当 Master 的事件到达 Slave 时，被认为是两个不同的事件。

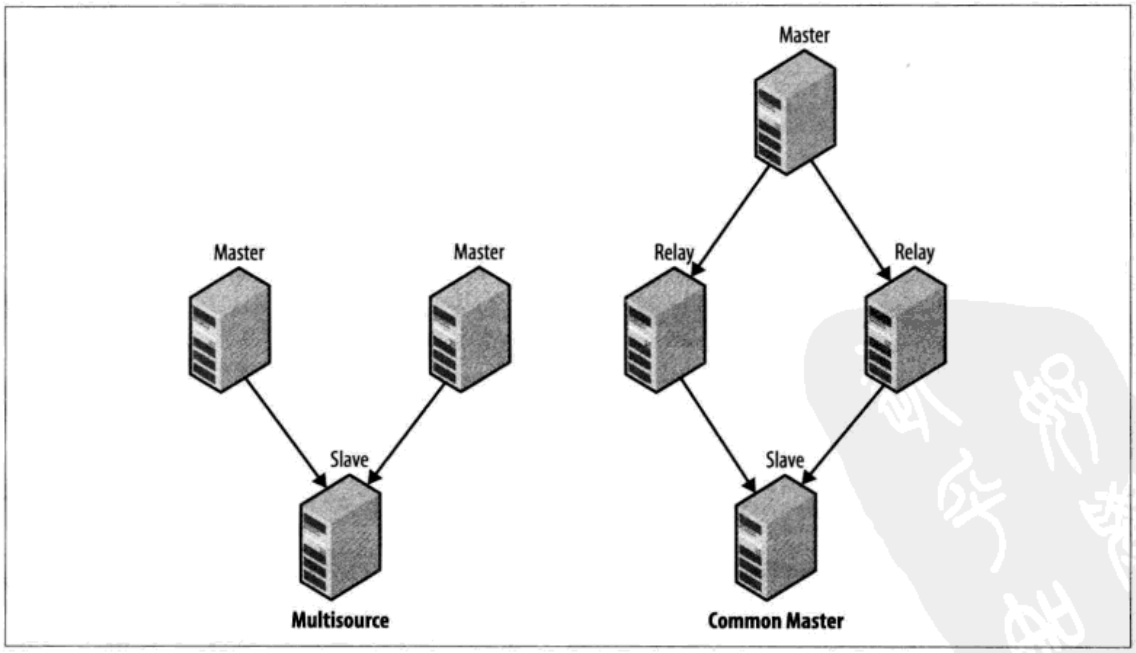


图6-6：两种类型的冲突



不需要明确配置：如果切换过程中复制流重叠，那么从一个中继切换到另一个中继时会不经意间发生这种情况。所以，需要保证所有事件都在队列中——Slave 以及 Master 和 Slave 之间的所有中继上的所有事件，在切换到另一个 Master 之前已经复制到 Slave 上了。

正确处理切换可以避免冲突（在多个数据源的情况下），确保所有更新已经完成，从而没有发生冲突的可能。典型的实现方法是，更新不同的数据库，但也可以将同一张表的不同行的更新分配到不同的服务器执行。

尽管目前 MySQL 不支持同时从多个源复制，但可以近似实现：将 Slave 在多个 Master 之间切换，轮流从其中一个 Master 定期复制，称为轮盘多源复制（*round-robin multisource replication*）。对某些应用来说这种方法很有用，比如从多个源聚集数据做报表。这时，将每个 Master 的写操作存在各自的数据库、表或分区中，自然地将数据分离。由于没有冲突的风险，所以可使用多源复制。

227

图 6-7 给出了以轮盘的方式将三个 Master 复制到 Slave 的例子，有一个客户端专门处理 Master 之间的切换。轮盘多源复制的过程如下。

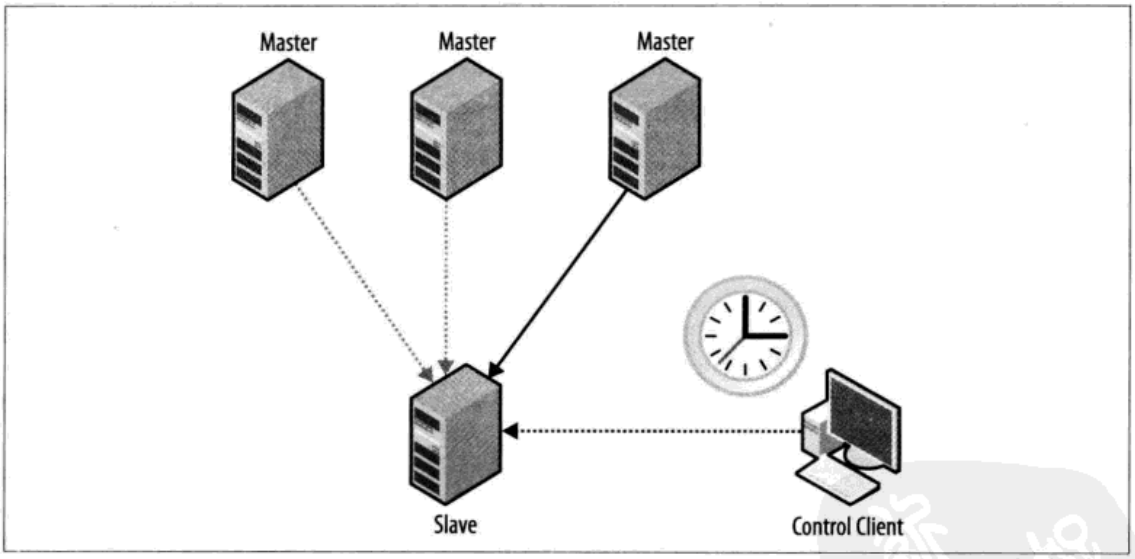


图6-7：以轮盘的方式将三个Master复制到Slave的例子

1. 将 Slave 配置为从一个 Master 复制，我们称这个 Master 为当前 Master。
2. 设置 Slave 复制的时间段。Slave 从当前 Master 中读取变化，然后当负责处理切换的客户端休眠时应用这些改变。

3. 使用 STOP SLAVE IO_THREAD 停止 Slave 的 I/O 线程。

4. 等待，直到中继日志为空。

5. 使用 STOP SLAVE SQL_THREAD 停止 SQL 线程。CHANGE MASTER 要求两个线程都停止。

6. 保存当前 Master 的 Slave 位置，存储 SHOW SLAVESTATUS 命令的输出结果 Exec_Master_Log_Pos 和 Relay_Master_Log_File 的值。

7. 将 Slave 的复制按顺序转到下一个 Master 上，利用之前保存的位置，并使用 CHANGE MASTER 命令建立连接。

8. 使用 START SLAVE 重启 Slave 线程。 228

9. 重复第 2 ~ 8 步。

注意第 3 ~ 5 步，我们先停止 I/O 线程，然后停止 SQL 线程。之所以要停止这两个线程，而不仅仅是停止 Slave 上的复制，原因是 SQL 线程可能拖慢进度（通常都会），所以，如果我们停止了这两个线程，则中继日志中会有大量事件被丢弃。如果只关心执行，比如说，一分钟内执行所有 Master 的事务，且不用关心丢弃的其他事件，那么你只需要停止复制，而不用执行第 3 ~ 5 步。但是，这个过程仍是对的，因为丢弃的事件将会在下一轮中重新读取。

当然，这个过程可以自动化，使用一个单独的客户端连接和 MySQL Replicant 库，如例 6-17 所示。使用 itertools 模块的 cycle 函数，可以轮流从一组 Master 中重复读取数据。

例6-17：轮盘多源复制的Python脚本

```
import itertools

position = {}
def round_robin_multi_master(slave, masters):
    current = masters[0]
    for master in itertools.cycle(masters):
        slave.sql("STOP SLAVE IO_THREAD");
        mysqlrep.wait_for_empty_relay_log(slave)
        slave.sql("STOP SLAVE SQL_THREAD");
        position[current.name] = mysqlrep.fetch_slave_position(slave)
        slave.change_master(position[current.name])
        master.sql("START SLAVE")
        current = master
        sleep(60)               # Sleep 1 minute
```

229

基于行的复制

复制的主要目的是保证 Master 和 Slave 之间的同步，使它们拥有相同的数据。前面已经知道，复制提供了很多特性来保证 Master 和 Slave 上的结果尽可能一致，比如上下文事件、基于会话的 ID 等。

尽管如此，基于语句的复制仍然存在很多目前无法解决的问题：

- 本章前面讲过，如果 UPDATE、DELETE 或 INSERT 语句包含 LIMIT 从句，则执行期间的数据库崩溃可能带来问题。
- 如果非事务性语句执行期间出错，则 Master 和 Slave 之间不能保证一致性。
- 如果语句含有 UDF 函数调用，则无法保证 Slave 上使用的是相同的值。
- 如果语句含有任何不确定的函数调用，例如 USER,CURRENT_USER, CONNECTION_ID 等，可能会导致 Master 与 Slave 之间产生不一致的情况。
- 如果语句更新的两张表中含有自增字段，则无法保证正确性。因为 Slave 上只是将同一个 ID 复制后插到两张表中去，而 Master 则分别为每张表插入一个 ID。

上述情况下，最好复制即将插入表中的真实数据，这就是基于行的复制（*row-based replication*）。

基于行的复制不复制语句，而是将插入、删除或更新操作的各行进行复制。由于发给 Slave 的行与发给存储引擎的行相同，其中包含插入表中的真实数据，所以不用考虑 UDF，没有自增量计数器，也不用考虑语句的部分执行——只需简单地考虑数据本身即可。

基于行的复制解决了基于语句的复制无法完成的事情。但要注意它们之间的区别。

选择基于语句的复制还是基于行的复制，需要考虑以下问题：

- 语句是否更新大量的行，还是通常只改变或插入少量行？
230
 如果语句改变大量的行，基于语句的复制执行更快。但是由于语句也在 Slave 上执行，所以并不总是这样。如果语句的优化和执行计划很复杂，这时可能基于行的复制更快，因为寻找行的逻辑快得多。
 如果语句只改变或插入少量行，则基于行的复制更快，因为不需要解析，所有的处理都直接交给存储引擎。
- 是否需要知道执行了哪些语句？至少可以说，基于行的复制的事件处理很难解码。而在基于语句的复制中，语句被写入二进制日志中，因此可以直接读取。
- 基于语句的复制的复制模型很简单：只要在 Slave 上执行相同的语句即可。这种技术应用已久，很多 DBA 对其较熟悉。而基于行的复制则相对较新，如果复制过程

出现故障，可能难以解决。

- 如果 Master 和 Slave 上数据不同，执行语句的结果也会不同。有时是故意的（这时应该使用基于语句的复制），但有时是无意的，可以通过基于行的复制来避免这一情况。

基于行的复制和基于语句的复制提供了不同的解决方法。前面已经讨论过基于语句的复制，本章将介绍如何使用基于行的复制。

基于行的复制参数

配置基于行的复制需要以下参数：

- `binlog-format`

 `binlog-format` 参数可以设置为下面几种模式：

 - `STATEMENT`

 所有语句都使用传统的基于语句的复制。

 - `ROW`

 所有插入或改变（即数据操纵语言 DML 语句）数据的语句都使用基于行的复制。但是，创建表或其他更改模式（schema）（即数据定义语言 DDL 语句）的语句仍使用基于记录的复制。

 - `MIXED`

 这是基于语句的复制的安全版本，MySQL 5.1 版本推荐使用此模式。在混合模式中，服务器将语句写入二进制日志中，如果语句不安全（根据前面介绍的判断标准），则切换为基于行的复制。

 231

 该变量还可作为全局服务器变量和会话变量。开始新的会话时，全局变量的值被复制到会话变量中，然后由会话变量决定如何将语句写入二进制日志。

- `binlog-max-row-event-size`

 该参数用于指定何时开始下一个包含行的事件。由于处理时事件完全被读入内存，该参数提供了控制事件大小的粗略方法，保证处理行时不会消耗过多的内存。

混合模式的复制

MySQL 5.1 版本推荐使用混合模式的复制，但 `binlog-format` 参数默认值为 `STATEMENT`。看上去很奇怪，但这样却避免了从 5.0 或早期版本升级带来的问题。因为旧版本没有基于行的复制，只能使用基于语句的复制，而且 MySQL 开发者也不想突然切换服务器。如果服务器升级时突然开始发送基于行的复制事件，那么部署将变得一团乱。为了减少升级需要考虑的因素，该参数的默认值仍为 `STATEMENT`。

但是，如果你用的是 MySQL 版本 5.1，就会发现 binlog-format 参数的值为 MIXED。

混合模式复制背后的原理很简单：正常情况下使用基于语句的复制，对不安全的语句切换为基于行的复制。我们探讨了导致这种切换的各种可能的问题及其原因。总的来说，当出现以下情况时，混合模式需要切换到基于行的复制：

- 该语句调用了：
 - UUID 函数；
 - 用户自定义函数；
 - CURRENT_USER 或 USER 函数；
 - LOAD_FILE 函数。
- 同一个语句更改了两张或更多包含 AUTO_INCREMENT 列的表。
- 语句中使用了服务器变量。
- 存储引擎不允许使用基于语句的复制，例如，MySQL Cluster 引擎。

232 当然，以上所列的情况并不完整：随着新的不安全因素被发现，这个列表也在不断扩展。更完整精确的列表参见在线 MySQL 参考手册（*http://dev.mysql.com/doc/refman/5.1/en/binary-log-mixed.html*）。

处理基于行复制的事件

在基于语句的复制中，语句由单个 Query 事件处理。但是，由于各个语句改变的行数目很大，基于行的复制处理方式不同，因此每个语句需要多个事件。

处理基于行的事件需要引入四个新的事件：

- Table_map 事件
 Table_map 事件将 Master 上的表 ID 映射到表名（包括数据库名）及其列的基本信息。表信息不包括列名，仅包含列类型。因为基于行的复制与位置相关——Master 上的各列与 Slave 表的列位置完全相同。
- Write_rows, Delete_rows 和 Update_rows 事件
 无论何时插入、删除和更新行，都会分别产生这三个事件。这意味着单个语句可以产生多个事件。
 除了行以外，这些事件还包含前面 Table_map 事件中的表 ID，以及一个或多个列位图（*column bitmap*）指定事件所涉及的列。这样日志只需记录那些发生改变的列，或者插入、删除或更新操作正确定位所需的列，从而节省了存储空间。
 目前，只有 MySQL Cluster 引擎在日志中使用参数来限制发送列的数目。

只要语句执行就会写入二进制日志，产生一个 Table_map 事件序列和一个行事件序列。

给语句的最后一个事件加上一个特殊标记，表明这是语句的最后一个事件。

例 6-18 显示了语句的执行及其事件。本例中，我们忽略了事件的格式化描述。

例6-18：INSERT语句的执行及其事件

```
master> BEGIN;
Query OK, 0 rows affected (0.00 sec)

master> INSERT INTO t1 VALUES (1),(2),(3),(4);
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0

master> INSERT INTO t1 VALUES (5),(6),(7),(8);
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0

master> COMMIT;
Query OK, 0 rows affected (0.00 sec)

master> SHOW BINLOG EVENTS IN 'master-bin.000053' FROM 106\G
*************************** 1. row ***************************
   Log_name: master-bin.000054
        Pos: 106
 Event_type: Query
  Server_id: 1
End_log_pos: 174
       Info: BEGIN
*************************** 2. row ***************************
   Log_name: master-bin.000054
        Pos: 174
 Event_type: Table_map
  Server_id: 1
End_log_pos: 215
       Info: table_id: 18 (test.t1)
*************************** 3. row ***************************
   Log_name: master-bin.000054
        Pos: 215
 Event_type: Write_rows
  Server_id: 1
End_log_pos: 264
       Info: table_id: 18 flags: STMT_END_F
*************************** 4. row ***************************
   Log_name: master-bin.000054
        Pos: 264
 Event_type: Table_map
  Server_id: 1
End_log_pos: 305
       Info: table_id: 18 (test.t1)
*************************** 5. row ***************************
```

233

```
   Log_name: master-bin.000054
        Pos: 305
 Event_type: Write_rows
  Server_id: 1
End_log_pos: 354
       Info: table_id: 18 flags: STMT_END_F
*************************** 6. row ***************************
   Log_name: master-bin.000054
        Pos: 354
 Event_type: Xid
  Server_id: 1
End_log_pos: 381
       Info: COMMIT /* xid=23 */
6 rows in set (0.00 sec)
```

234 这个例子向二进制日志写入了两条语句。每个语句都以 Table_map 事件开始，然后 Write_rows 事件将写入四行记录。

为行事件设置 statement-end 标志来结束语句。由于语句包含在事务内，所以它们被封装在包含 BEGEIN 和 COMMIT 语句的 Query 事件内。

行事件的大小由 binlog-row-event-max-size 参数来控制，该参数规定了二进制日志中字节数的阈值，但没有指定行事件的最大值：如果行中包含的字节数超过了 binlog-row-event-max-size，可能会有一个更大的 binlog 行事件。

表映射事件

前面已经讲过，表映射事件（Table_map 事件）将表名映射为行事件的标识符，但这并不是表映射事件的唯一功能，它还包含 Master 表的字段信息。Slave 可以利用这些信息检查 Slave 表的基本结构，然后与 Master 比较，保证它们足够一致再进行复制。

图 6-8 给出了表映射事件的基本结构。通用头（common header）（所有复制事件都有这个头）包含事件的基本信息。通用头之后是 post 头（post header），含有表映射事件的特殊信息。图 6-8 中的大部分字段都可以直接从名字知道其功能，不过我们还需要进一步看看如何表示各个字段的类型。

下面给出了几种字段类型：

- 字段类型数组
 这是关于所有字段的基本类型的数组，指明某字段是整型、字符串类型、数字类型或者其他可用类型，但并没有提供列类型的其他参数。例如，如果某字段的类型是 CHAR(5)，那么这个数组大小为 254（表示一个字符串的常数），而字符串的长度（这里是 5）则存储在字段元数据中（下面即将介绍）。

- 空比特数组
 表明每个字段是否为空的比特数组。
- 字段元数据
 字段的元数据数组，扩展了字段类型数组的详细信息。每个字段可用的元数据取决于该字段的类型。例如，DECIMAL 字段的元数据存储了精度和整数及小数部分，而 VARCHAR 类型存储了字段的最大长度。

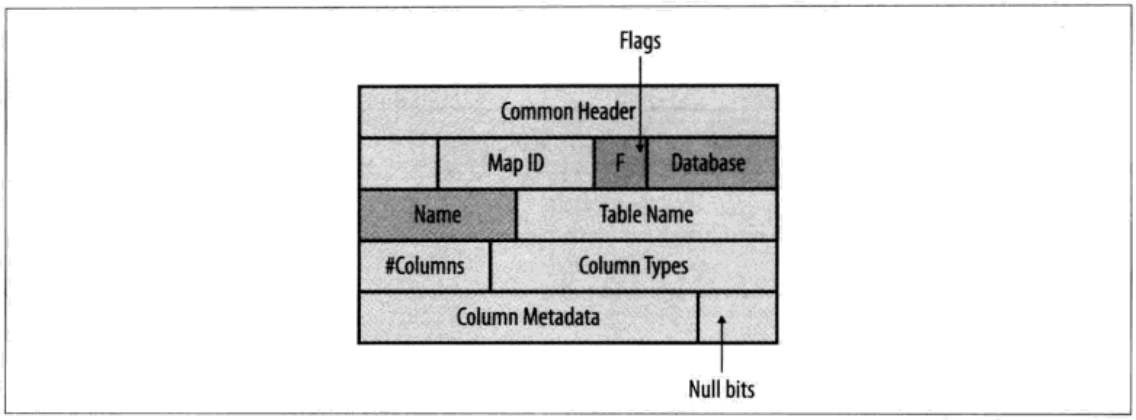


图6-8：表映射事件的基本结构



结合这三个数组中的数据，可以推断出字段的类型。

不是所有的类型信息都存储在数组中。以下两种特殊的情况下，Master 和 Slave 无法确定字段类型：

- 没有指明整型字段的符号。这意味着 Slave 在检查表时无法判断某字段是否有符号。
- 没有指明字符串类型的字符集。这意味着不支持不同字符集之间的复制，而且可能导致奇怪的结果，因为字节未经检查或转换就被插入字段中。

行事件的结构

图 6-9 显示了行事件的结构。不同的事件类型(写、删除或更新)，行事件的结构稍有不同。

除了表 ID（即前面表映射事件中的表 ID）以外，该事件还包含以下字段：

- 表宽
 Master 表的宽度。其编码方式与客户端协议相同，可以是一个或两个字节，大部分情况下是一个字节。
- 字段位图
 作为事件的有效载荷的一部分被发送的那些字段。该信息允许 Master 发送一行的某

些特定字段。字段位图有两种类型：一种用于前映像（*before image*），另一种用于后映像（*after image*）。前映像用于删除和更新，而后映像用于写（插入）和更新，要获得更多信息，请看表 6-1。

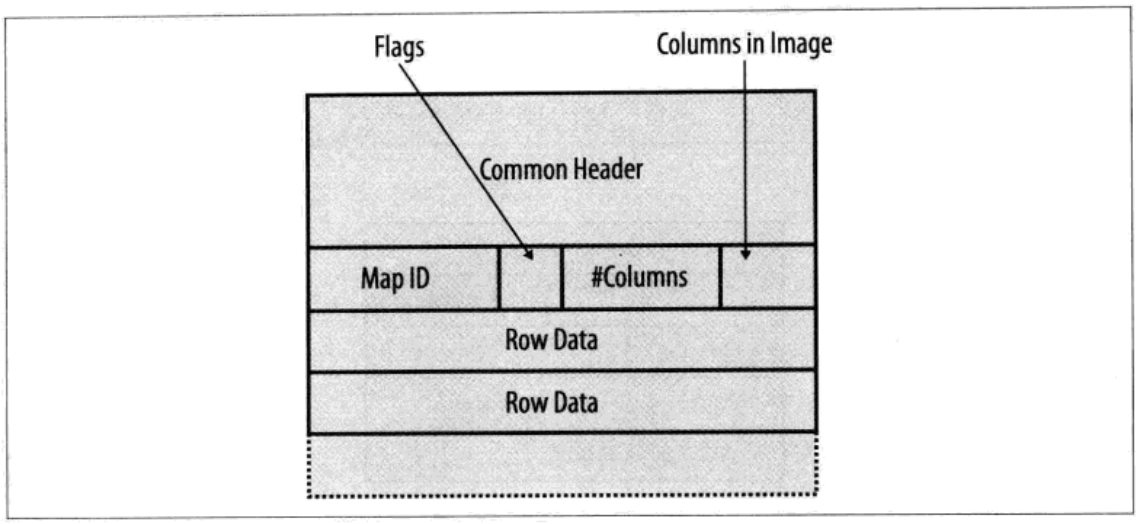


图6-9：行事件结构

表6-1：行事件及映像

前像	后像	事件
无	要插入的行	写入行
要删除的行	无	删除行
更新前的列值	更新后的列值	更新行

事件的执行

因为多个事件可以表示 Master 上执行的单个语句，所以 Slave 需要保存状态信息，从而当并发线程更新同一张表时，保证行事件的正确执行。回想一下，二进制日志中的每个语句都以一个或多个表映射事件开始，后面是一个或多个类型相同的行事件。二进制日志的语句处理步骤如下：

1. 从中继日志中读取各个事件。
2. 如果是表映射事件，SQL 线程将提取表信息，并保存 Master 表的定义。
3. 出现第一个行事件时，锁定列表中的所有表。
4. 线程检查每张表在 Master 上的定义是否与 Slave 上的定义一致。

5. 如果不一致，线程报错，并停止 Slave 复制。

6. 执行后面的行事件处理步骤，直到线程读取到最后一个事件——即带有语句结束标志的事件。

与 Master 上的语句执行过程类似，这个过程也需要正确锁定 Slave 表。第三步中的所有表都被锁定，然后第四步进行检查。如果检查表定义之前不锁定表，Slave 线程可能会更改表定义，从而导致行事件的应用程序失效。 237

根据事件类型不同，各个行事件对行的处理不同。对于 `Delete_rows` 和 `Write_rows` 事件，每行表示一个改变。而 `Update_rows` 事件有两行（一行用来正确定位需要更新的行，另一行存储新的行值），所以该事件含有偶数个行，每两行表示一次更新。

拥有前映像（before image）的事件需要经过查找后正确定位到需要操作的行：`Delete_rows` 事件删除行，而 `Update_rows` 事件更改行。按照优先级递减的顺序，这些查找操作包括：

- 主键查询
 如果 Slave 表有主键，就使用主键进行查询操作。这是最快的方法。
- 索引扫描
 如果表中没有定义主键，但定义了索引，就使用索引定位到需要改变的行。扫描索引中的所有行，然后与 Master 行进行比较。
 如果找到了匹配的行，则执行 `Delete_rows` 或 `Update_rows` 操作。否则，Slave 停止复制，报告“找不到正确的行”的错误。
- 表扫描
 如果表上既没有主键也没有索引，则使用全表扫描。
 与索引扫描一样，每行都会扫描，然后与 Master 行比较，接着对匹配的行执行删除或更新操作。

由于使用的是 Slave 上的索引或主键来定位正确的行进行 delete 或 update，而不是 Master，因此需要注意以下几点：

- 如果 Slave 表有主键，查询将很快；如果没有，则必须进行全表扫描或索引扫描，相对较慢。
- Master 和 Slave 上的索引可能不同。



无论采用基于行的复制还是基于语句的复制，最好都使用主键来复制表。 238

由于基于语句的复制实际上执行了每条语句，所以主键的更新和删除也极大地加速了基于语句的复制。

事件和触发器

基于语句的复制和基于行的复制，对事件和触发器的执行过程不同。对于事件来说，唯一的区别就是基于行的复制产生行事件而不是查询事件。而触发器的区别更大。

第 3 章中讨论过，基于语句的复制将触发器的定义复制到 Slave，所以执行带有触发器的语句时，Slave 上也会执行触发器。

而对基于行的复制来说，无论行如何改变——无论变化来自触发器、存储过程、事件或者直接来自于语句本身，由于触发器更新的行已经被复制到 Slave，所以触发器不需要在 Slave 上再次执行。实际上，如果在 Slave 上执行触发器会导致不正确的结果。

例 6-19 定义了一个带有触发器的表。

例6-19：表及触发器的定义

```
CREATE TABLE log (
    number INT AUTO_INCREMENT PRIMARY KEY,
    user CHAR(64),
    brief TEXT
);

CREATE TABLE user (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email CHAR(64),
    password CHAR(64)
);

CREATE TRIGGER tr_update_user AFTER UPDATE ON user FOR EACH ROW
    INSERT INTO log SET
       user = NEW.email,
       brief = CONCAT("Changed password from '",
                      OLD.password, "' to '",
                      NEW.password, "'");

CREATE TRIGGER tr_insert_user AFTER INSERT ON user FOR EACH ROW
    INSERT INTO log SET
        user = NEW.email,
        brief = CONCAT("User '", NEW.email, "' added");
```

239

定义好表和触发器后，顺序执行下面的语句。

```
master> INSERT INTO user(email,password) VALUES ('mats@example.com',
'xyzzy');
Query OK, 1 row affected (0.05 sec)

master> UPDATE user SET password = 'secret' WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.01 sec)
```

```
Rows matched: 1  Changed: 1  Warnings: 0

master> SELECT * FROM log;
+--------+--------------+----------------------------------------+
| number | user         | brief                                  |
+--------+--------------+----------------------------------------+
|      1 | mats@sun.com | User 'mats@example.com' added          |
|      2 | mats@sun.com | Changed password from 'xyzzy' to 'secret' |
+--------+--------------+----------------------------------------+
2 rows in set (0.00 sec)
```

当然，这么做不太安全，但至少可以说明情况。那么，使用基于行的复制时，二进制日志将如何反映这些变化呢？

```
master> SHOW BINLOG EVENTS IN 'mysqld1-bin.000054' FROM 2180;
+----------------+----+-----------+---------+-----------+-----------------------+
|Log_name        |Pos |Event_type |Server_id|End_log_pos|Info
|
+----------------+----+-----------+---------+-----------+-----------------------+
|master-bin.000054|2180|Query      |        1|       2248|BEGIN                  |
|master-bin.000054|2248|Table_map  |        1|       2297|table_id: 24 (test.user)|
|master-bin.000054|2297|Table_map  |        1|       2344|table_id: 26 (test.log) |
|master-bin.000054|2344|Write_rows |        1|       2397|table_id: 24           |
|master-bin.000054|2397|Write_rows |        1|       2471|table_id: 26 flags:    |
|                 |    |           |         |           |  STMT_END_F           |
|master-bin.000054|2471|Query      |        1|       2540|COMMIT                 |
|master-bin.000054|2540|Query      |        1|       2608|BEGIN                  |
|master-bin.000054|2608|Table_map  |        1|       2657|table_id: 24 (test.user)  |
|master-bin.000054|2657|Table_map  |        1|       2704|table_id: 26 (test.log)   |
|master-bin.000054|2704|Update_rows|        1|       2783|table_id: 24           |
|master-bin.000054|2783|Write_rows |        1|       2873|table_id: 26 flags:    |
|                 |    |           |         |           |   STMT_END_F          |
|master-bin.000054|2873|Query      |        1|       2942|COMMIT                 |
+----------------+----+-----------+---------+-----------+-----------------------+
12 rows in set (0.00 sec)
```

可见，每个语句都是一个只包含单个语句的事务。语句更改两张表（*test.user* 表和 *test.log* 表），因此二进制日志中，语句的开始处有两个表映射事件。事件复制到 Slave 后直接执行，不考虑触发器，从而避免了 Slave 表再次执行触发器。

过滤

240

基于语句的复制和基于行的复制对过滤的处理也不相同。回忆一下第 3 章，基于语句的复制是在整个语句上完成过滤的（或者所有语句都执行，或者所有语句都不执行），因为只执行部分语句是不可能的。数据过滤使用当前数据库而不是正在操作的表所在的数据库。

而基于行的复制提供了更多选择。因为表的每一行都会复制，所以可以过滤正在更新的真实表，甚至可以基于任意条件过滤行。因此，基于行的复制的过滤处理是基于真正发生变化的表，而不是语句的当前数据库。

考虑一下，如果将 Slave 配置改为忽略 *ignore_me* 数据库，那么如何过滤呢？对于基于语句的复制和基于行的复制，执行下面的语句分别会产生什么结果？

```
USE test; INSERT INTO ignore_me.t1 VALUES (1),(2);
```

基于语句的复制执行该语句，而基于行的复制则忽略表 *t1* 上的改变，因为 *ignore_me* 数据库已被忽略。

接着执行下面的多表更新语句又将怎样呢？

```
USE test; UPDATE ignore_me.t1, test.t2 SET t1.a = 3, t2.a = 4 WHERE t1.a =
t2.a;
```

基于语句的复制执行该语句，认为 *ignore_me.t1* 表存在（实际上该表并不存在，因为数据库已被忽略）并更新 *ignore_me.t1* 表和 *test.t2* 表。而基于行的复制只更新 *test.t2* 表。

语句的部分执行

注意，如果不考虑失效、崩溃及非确定性行为，通常基于语句的复制性能很好。最坏的情况下，失效或崩溃常常导致语句部分执行。

如果 UPDATE，DELETE 或 INSERT 语句影响的行数被人为地限定，也会导致语句部分执行。例如，使用 LIMIT 语句或者表是非事务性的，比如，重复键错误导致执行中止，这时语句将部分执行。

这些情况下，语句所描述的变化只是部分应用到某个初始的行集。由于 Master 和 Slave 上这些行的顺序不同，所以语句应用的行集也不同。

241

MyISAM 维护了所有行的顺序，如果发生部分执行，可以帮助你更新相同的行集。但不幸的是，事实并不是这样。如果 Slave 通过逻辑备份或从备份中恢复的方式从 Master 克隆数据，行的插入顺序可能改变。

通常使用 ORDER BY 从句可以解决这个问题，但这也不完全安全，因为仍有可能由于崩溃而导致语句部分执行。

小结

本章总结了关于MySQL复制的一系列章节，讨论了高级复制的相关内容，比如如何更加健壮地将Slave提升为Master，介绍了一旦数据库崩溃如何避免损坏的技巧和技术，解释了多源复制的配置及其需要考虑的问题，最后详细介绍了基于行的复制。

在下一章中，我们将学习如何构建健壮的数据中心，包括监控、存储引擎的性能调优、复制。

Joel吃完午饭回来，在大厅的走廊上碰见了老板，“您好，Summerson先生。”

“你好啊，Joel。”

“您看过我的报告了吗？”

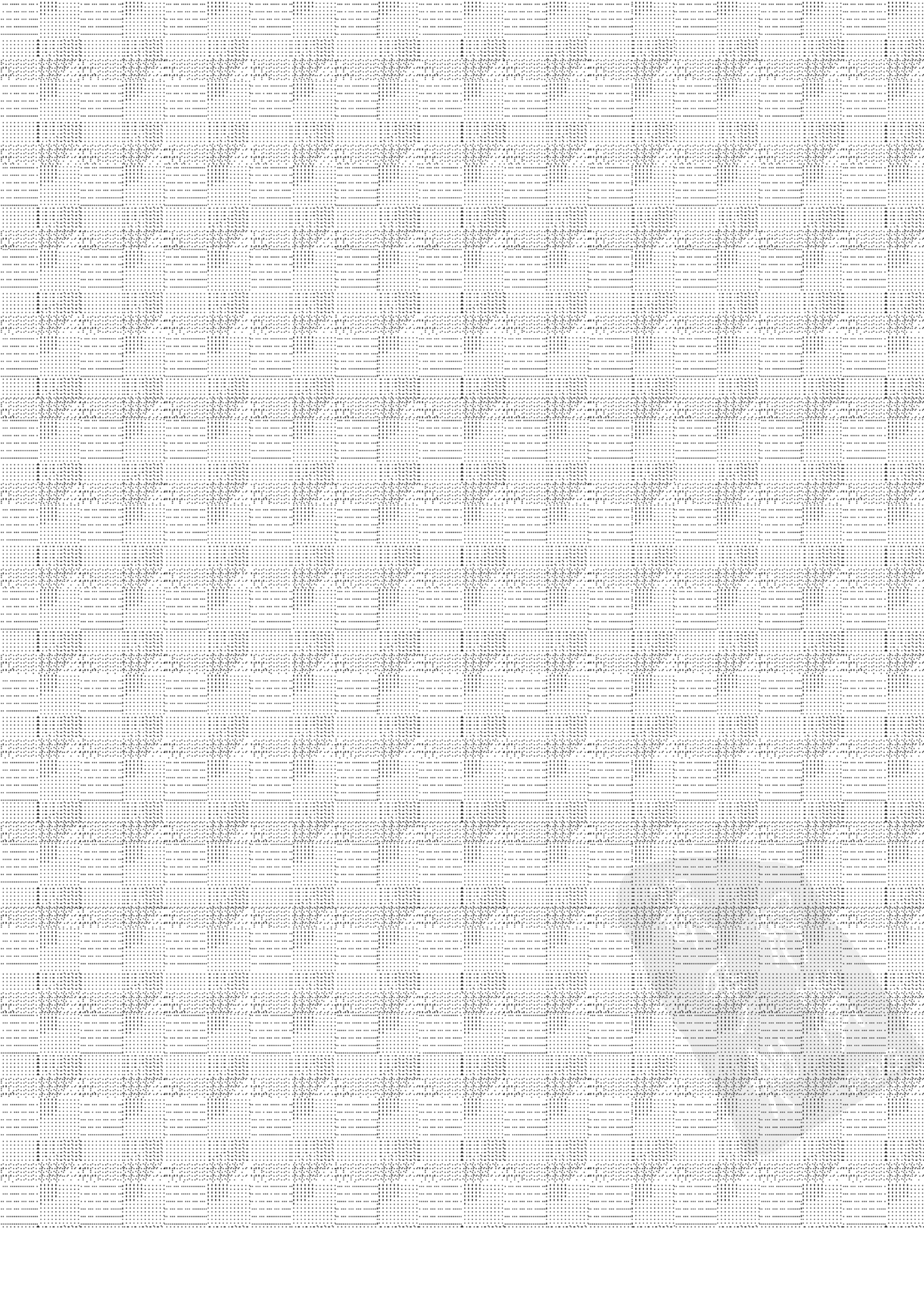
“看过了，Joel。做得不错嘛。我已经分发给其他几个部门，看看他们有什么意见。我想把它加到我们的SOP手册中。”

Joel心想SOP意味着标准的操作流程。

“我已经叫他们看过以后把意见发给你。要想加到SOP中可能还需要一些语言上的润饰，我想你肯定行。”

“谢谢，先生。”

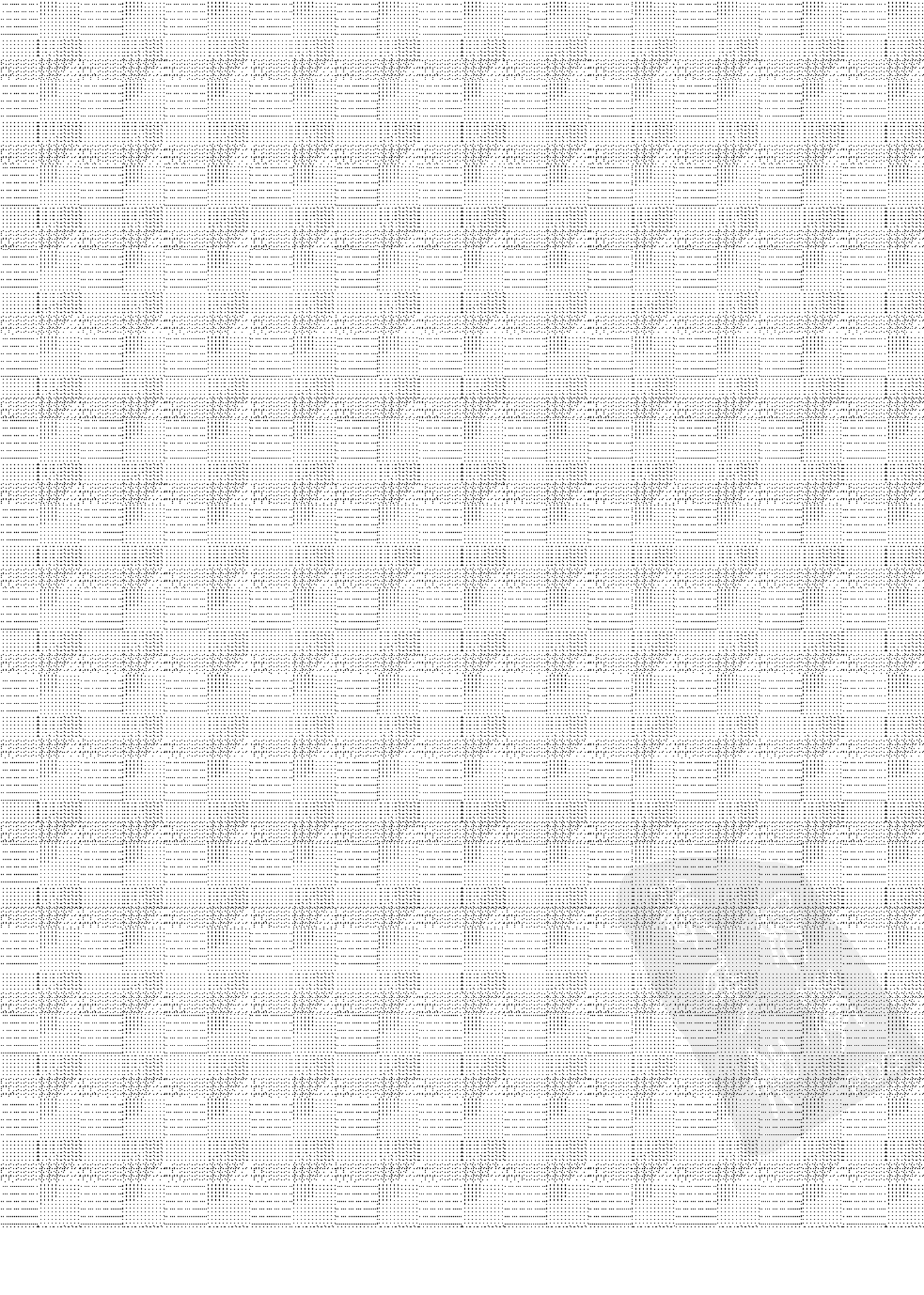
Summerson先生点点头，拍了拍Joel的肩膀，然后走了。



第二部分

监控和灾难恢复

现在你有一个复杂的多服务器系统，希望它能满足你的网站需求，且必须能正确地掌控它。本书的这一部分介绍了监控问题，还介绍了一些性能、备份和处理偶尔发生的不可避免的故障的其他方面的问题。



第7章 监控入门

Joel 将半脱脂拿铁咖啡、水果杯和奶酪糕点放到桌子上，并对着那些等着他的营养品笑了。自从在上班的路上发现了一家高档购物中心后，他的早餐就变得相当有创意了。

他一边打开拿铁的盖子，一边打开显示器等待邮件系统检索信息，然后浏览着这些邮件的标题，并希望没有收到老板发来的新消息。这时，他注意到有一些从用户那里发来的关于性能问题的新消息。

Joel 打开这些信息并浏览其内容。当他读到有些人抱怨查询数据库系统的应用程序的响应时间太长时，嘴里嘀咕着："嗯，我想一定是什么地方出错了。"

他边打开糕点边思索着是什么导致出现这些问题。"系统昨天还是好好的。"他想。在吸了几口拿铁后，他回想起了在大学实验室工作时读到的一些关于性能监控的知识。

Joel 吃完糕点后拿起《高可用 MySQL》这本书，并自言自语："这本书中有我想要查看的内容。"

如何发现服务器运行不佳的状况？等到用户告诉你系统有问题时，这些问题可能已经存在有一段时间了。如果问题长时间得不到解决，将会使系统诊断和修复过程复杂化。

本章将使用各种系统提供的基本工具，在操作系统层面对 MySQL 监控进行测试，我们从这里开始介绍是因为系统服务或应用程序总是依赖于操作系统和其本身硬件的性能。如果操作系统性能很差，其上安装的数据库系统或应用程序的性能也好不到哪里去。

我们首先考察为什么要使用监控系统，再看看在主流操作系统上运行的基本监控任务，

246 并讨论监控系统如何使预防性维护工作变得更轻松。一旦你掌握了这些技巧，就可以更加了解你的数据库系统。下一章将关注监控 MySQL 服务器的更多详情，并介绍一些解决常见性能问题的使用指南。

监控方法

当我们想到“监控”时，通常会联想到一些用于监测问题的早期预警系统。然而，监控（作为动词）这个词是指“在不影响操作或运行环境的前提下，用仪器观察、记录或检测操作或环境”（*http://www.dictionary.com*）。这些早期的预警系统采用自动采样和预报系统相结合的方式进行预警。

Linux 和 UNIX 操作系统非常复杂，且有许多影响主要和次要系统活动的所有行为的参数。优化这些操作系统的性能可以说是一门艺术而非科技。与一些桌面操作系统不同，Linux 和 UNIX（及其变种）既不隐藏系统优化工具，也不限制你对系统的优化。而一些桌面操作系统，如 Mac OS X 和 Windows，将系统的许多基础机制隐藏在用户非常友好的可视化界面后面。

例如，Mac OS X 操作系统是一个非常优雅的且运行流畅的操作系统，在正常情况下它只需要一点点或根本不需要用户的注意。然而，在以下章节你将看到，Mac OS X 操作系统提供许多先进的监控工具，这些工具可以帮你优化 Mac OS X 操作系统。

Windows 操作系统有很多版本，目前最新的版本是 Windows 7。幸运的是，这些版本中的大部分都包含相同的监控工具，用户可以使用这些监控工具优化系统以满足特定需求。虽然 Windows 不如 Mac OS X 操作系统那样讨好，但是它提供了许多用户可访问的优化选项。

主要有三类系统监控：系统性能、应用程序性能和安全性。你可能因为一些特殊原因而开始进行系统监控，但是一般情况下任何监控任务都可以归为这三类中的一类。

每类都使用不同的工具集（有一些重叠）且监控对象也不相同。例如，监控系统性能可以确保系统能够在最高效率下运行，监控应用程序的性能是为了确保单个应用程序能够以最高效率执行，而监控安全性却是为了确保系统被保护在最安全的保护模式下。

监控 MySQL 服务器类似于监控应用程序。这是因为 MySQL 就像大多数的数据库系统一样，需要测量许多与操作系统关系不大或无关的变量和状态指标。但是，数据库系统
247 非常容易受到主机操作系统的影响，所以在进行数据库系统的问题诊断前，确保主机操作系统运行良好是很重要的。

因为我们的目的是监控 MySQL 系统以确保数据库系统能够以最高的效率执行，所以以

下各节将讨论操作系统的性能监控。至于安全性监控，将在其他章节详细介绍。

监控的好处

监控方法有两种：主动监控和被动监控。你可能希望通过监控来确保一切都没有改变（没有性能下降，不存在安全漏洞），或者查出是哪些地方发生了改变或哪些地方出现了问题。主动监控是指通过监控系统确保一切都没有改变，而被动监控是指通过监控系统确定系统出错的地方。遗憾的是，大多数监控是被动监控，且被动监控是有些专业人士了解的唯一形式。只有极少数的 IT 专业人士有时间或资源进行主动监控。

不过，花时间主动监控系统可以减少大量的被动监控工作。例如，如果你的用户抱怨系统性能差（被动监控首要关注的问题），你没有办法知道该系统退化了多少，除非有以前的监控结果可以进行对比。记录这种监控结果被称为“形成系统的基准”。也就是说，你在低、正常和高负荷的环境下已经监控系统性能一段时间了。如果你持续频繁地进行采样，就可以确定在各种负荷情况下系统的典型性能。因此，当用户报告性能问题时，你可以对系统进行采样并将上报的结果与基准进行比较。如果你的历史数据足够详细，通常一眼就能发现系统的哪一部分已经发生改变了。

监控系统组件

在进行性能监控时，应该监控系统的四个基本组件：

- 处理器
 检查其利用率及达到了什么样的峰值。
- 内存
 检查内存被占用量，及其可用量。
- 磁盘
 检查磁盘空间可用量，磁盘空间是如何被占用的，还有哪些需求需要占用磁盘空间，还需要了解磁盘读取速度（响应时间）。
- 网络
 检查网络通信的吞吐量、延迟和错误率。

248

处理器

监控 CPU 以确保系统不存在失控进程，并确保 CPU 周期在各运行进程中平均分配。要做到这一点，方法之一是调用正在运行的进程列表，然后确定每个进程所占 CPU 的百分比。另外一个方法是检测系统进程的平均负载。大多数操作系统提供了 CPU 性能的多种视图。



进程是指在 L.inux 或 UNIX 操作系统上运行的一个工作单元。一个程序可能同时运行一个或多个进程。多线程应用程序，如 MySQL，通常以多进程的形式出现在操作系统中。

当 CPU 处在高性能负载下且争用激烈时，系统将运行缓慢，甚至出现死机情况。在这种情况下必须减少进程的数量或者减少那些消耗较多 CPU 时间的进程的 CPU 使用率。然而一定要监控 CPU，并确认系统问题确实是由 CPU 的高使用率导致的——系统运行缓慢更有可能是内存争用导致的，下一节将讨论这个问题。

CPU 超载的一些常见的解决方法是：

- 提供新服务器运行进程
 这当然是最好的办法，但是新服务器需要资金。有经验的系统管理员通常可以找到其他的方法去减少 CPU 的使用率，尤其是当组织更愿意花时间而不是花钱在这件事情上时。
- 删除不必要的进程
 大量系统的后台运行进程可能在某些场合有用，但大多数时候会使系统陷入瘫痪。然而，作为一个系统管理员，必须非常了解操作系统，以确定哪些进程是不必要的。
- 杀死失控的进程
 当性能问题间歇地或偶尔出现时，可能是由有缺陷的应用程序导致的，这些失控的进程往往是罪魁祸首。倘若你使用可控的或有序的方法不能够停止失控的进程，可能需要使用强制退出对话框或使用命令行强制性地终止这个进程。

249

- 优化应用程序
 有些应用程序时常会占用超过其实际需要的 CPU 时间或其他资源。设计糟糕的 SQL 语句经常拖累了数据库系统。
- 较低的进程优先级
 有些进程可以在后台作业运行，例如报表生成器，而且它们可以运行得更慢，以给互动进程腾出空间。
- 重新安排进程
 也许有些报表生成进程可以在系统负荷较低的夜间运行。

占用太多 CPU 时间的进程被称为 *CPU-bound* 或 *processor-bound*，这意味着它们不会为等待 I/O 暂停自己，也不能被交换出内存。

如果你发现 CPU 没有被争用，此时仅有少数几个进程在运行或者不存在消耗大量 CPU 时间的进程，那么此时很可能是其他地方导致性能问题的发生：如等待磁盘 I/O、内存不足、过度页交换等。

内存

监控内存是为了确保应用程序不要请求过多的内存，因为请求过多的内存会浪费系统管理内存的时间。操作系统从最初使用有限的随机存取存储器（RAM 或主内存），已经发展到使用磁盘存储器来存储未使用的部分或主内存页面，这种技术被称为分页或交换，可以存储悬停进程的内存，并在进程被激活时将被存储的内存恢复出来，这样系统在同一时间内比主内存系统负担的进程更多。虽然内存块在内存与磁盘间交换的代价相对较高（与直接访问主内存相比，比较耗时），但是现代操作系统在这方面能够做到很快，那么这种缺陷就不是问题了，除非它无法使处理器和磁盘的运行速度跟上需求。

然而操作系统可能定期回收较高水平的交换内存。一定要测量一段时间内的内存使用率，确保不存在常规清除操作。

在高分页期，内存不足有可能是因为失控进程占用太多的内存或者系统运行了过多的进程以至于占用过多的内存。这种高分页被称为 *thrashing*，与 CPU 资源争用类似。消耗过多内存的进程被称为 *memory-bound*。

在处理内存性能问题时，自然想到的处理方法是增加更多的内存。虽然这样可以解决问题，但内存也有可能在各子系统间分配不均匀。 250

在这种情况下你有几件事情可以做，可以分配不同数量的内存给系统的组件（如内核或文件系统），或者分配不同数量的内存给允许进行内存调整的各种应用程序（如 MySQL），还可以更改子系统的分页优先级，这样可以使操作系统更早开始分页。



调整服务器内存子系统时要小心，请务必参考关于提高特定操作系统性能的文档或书籍。

如果在监控内存时发现系统的分页并不频繁，而系统性能仍存在问题，此时，性能问题可能跟其他的子系统有关。

磁盘

监控磁盘使用率是为了确保系统拥有足够的可用磁盘空间且拥有足够的 I/O 带宽，以使进程执行时不会出现明显的延时。可以使用 *per-process* 或 *overall transfer* 读取磁盘的传输速率来衡量以上情况。per-process 传输速率是指单个进程可以读写的数据量。overall transfer 传输速率是指读写磁盘数据的最大带宽。一些有多个磁盘控制器的系统可能单独衡量每个磁盘控制器的 overall transfer 传输速率。

如果有一个或更多的进程消耗过多的最大磁盘传输率，将会出现性能问题。这就像进程消耗太多的 CPU 周期一样，同样会对系统的其余进程产生不利的影响：它将“饿死”其他进程，迫使它们等待更长的磁盘访问时间。

消耗太多磁盘传输率的进程被称为 *disk-bound*，也就是说它访问磁盘的频率大于磁盘传输率能够提供的份额。如果能够减少 disk-bound 进程给 I/O 系统带来的压力，将会为其他进程腾出更多的带宽。

一种满足执行大量的磁盘 I/O 的进程需要的方法是增加文件系统的块大小，从而使大型传输更有效，这样还可减少由 disk-bound 进程带来的系统开销。然而，这可能会使其他进程运行得更慢。



当在只有一个控制器或磁盘的服务器上优化文件系统时需要谨慎行事。请务必参考提高具体操作系统性能的文档或书籍。

251 如果资源充足，那么可以添加新的磁盘控制器和磁盘阵列来处理磁盘争用，将其中一个 disk-bound 进程的相关数据移到新的磁盘控制器上。另一种处理磁盘争的方法是将 disk-bound 进程移到另外一个使用率较低的服务器上。最后一种方法是在某些情况下，通过升级磁盘系统来使系统运行更快，从而增加磁盘的带宽。

至于先从哪里开始优化或者哪种优化方法是最好的，有不同观点。但是我们认为：

- 如果需要运行大量的进程，就需要扩大磁盘传输速率或将进程分布在不同的磁盘整列或系统上。
- 如果需要运行少数几个数据访问量大的进程，就需要通过增加文件系统的块大小来增加单个进程的传输速率。

可能需要在这两个解决方案之间取得平衡，通过将有些进程移到其他系统上的方法来满足独特的混合进程（mix of processes）。

网络子系统

监控网络接口以确保系统拥有足够的带宽，并确保正在发送或接收的数据具有高质量。

有些进程试图读写的数据超过网络配置或硬件所允许的范围，以至于它们消耗了过多的网络带宽，这样的进程被称为 *network-bound*。这类进程为了避免自己发生延时而阻止其他进程访问充足的网络带宽。

网络带宽问题通常表现为完全占用网络接口的最大带宽，可以通过给不同进程分配特定

网络端口的方式来解决这个问题。

网络数据质量问题通常表现为在网络接口上遭遇大量错误。幸运的是，操作系统和数据传输应用程序通常采用 *checksumming* 或其他一些算法来检测这类错误，但是重发将给网络和操作系统带来沉重负担。解决这个问题可能需要将一些应用程序移到同一网络的其他系统上，或者可能需要安装额外的网卡，而且在改变网络硬件、重新配置网络协议或者将系统移动到网络中不同的子网后，通常需要进行诊断分析。



当提起进程时你可能听到 *I/O-bound* 或 *I/O-starved* 这样的术语。这通常是指进程占用太多的磁盘或网络带宽。

监控方法

252

现代操作系统提供了获取以上四个子系统的状态信息的各自专用工具。这些工具大多是独立的且与其他工具至少无直接联系的应用程序。正如你将在接下来的章节看到的，这些工具凭自己的能力拥有强大的功能，但是需要大量精力去记录和分析它自己产生的所有数据。

幸运的是，大量的第三方监控方案可供大多数操作系统和数据库系统使用。下面是一些较出色的监控产品。最好让系统提供商推荐能够满足需求且能够保持与基础设施兼容的最佳监控方案。大多数供应商会提供系统监控工具供选择。

- *up.time*
 http://www.uptimesoftware.com/
- *Cacti*
 http://www.cacti.net/
- *KDE System Guard (KSysGuard)*
 http://docs.kde.org/stable/en/kdebase-workspace/ksysguard/index.html
- *Gnome System Monitor*
 http://library.gnome.org/users/gnome-system-monitor/
- *Nagios*
 http://www.nagios.org/
- *Sun Management Center*
 http://www.sun.com/software/products/sunmanagementcenter/index.xml
- *MySQL Enterprise Monitor*
 http://www.MySQL.com/products/enterprise/monitor.html



第 13 章将详细讨论 MySQL 企业版监控及自动监控和报告。

以下各节描述一些主流操作系统的内置监控工具。我们将更详细地研究一些 Linux 和 UNIX 命令。因为它们特别适合用于研究我们讨论过的性能问题和策略。同样，也将介绍 Mac OS X 和微软 Windows 上的监控工具。

Linux和UNIX监控

Linux 和 UNIX 上的数据库监控工具包括监控 CPU、内存、磁盘、网络、安全性和用户的监控工具。在传统的 UNIX 中，所有的核心工具从命令行运行，且大部分位于 *bin* 文件夹中或 *sbin* 文件夹中。表 7-1 罗列了我们找到的有用工具及其简单描述。

表7-1：Linux和UNIX系统监控工具

工具	描述
ps	显示系统上运行的进程列表
top	显示根据 CPU 使用率排序的活动进程
vmstat	显示内存、分页、块传输和 CPU 活动的相关信息
uptime	显示系统运行了多长时间。并显示用户登录数量和在 1 分钟、5 分钟和 15 分钟的内系统平均负荷量
free	显示内存使用率
iostat	显示平均磁盘活动和处理器负载情况
sar	显示系统活动报告。允许你收集和报告各种系统活动
pmap	显示各种进程分别占用内存的情况
mpstat	显示多处理器系统的 CPU 使用率
netstat	显示网络活动的相关信息
cron	可以让你安排进程执行的子系统。你可以安排这些实用程序的执行，故可以随着时间的推移定期收集统计信息，并可以在特定时间（如在高负载器或低负载期间）查看统计信息。



某些操作系统提供了额外的或可供选择的工具。参看你的操作系统文档上有关监控系统性能的其他工具。

正如表 7-1 所示，存在大量能提供潜在有用信息的实用工具。以下各节讨论一些比较流行的工具，并简要介绍如何使用它们识别出在上述章节中描述的那些问题。

进程活动

有几个命令提供运行在系统上的进程的相关信息，尤其是 top、iostat、mpstat 和 ps。

top命令

254

top 命令提供系统信息的摘要和按任务的 CPU 密集度排序的系统进程的动态视图。这些显示信息一般包含进程的信息，其中包括进程 ID、进程使用者、进程优先级、进程的 CPU 占用百分比和进程的耗时，当然，这个命令过去常常用于启动进程。然而，有些操作系统的报告会略有不同。top 命令可能是这组命令中最流行的实用工具，因为它提供了每隔几秒钟的系统快照。图 7-1 显示了 Linux（Ubuntu）系统在中等负载情况下运行 top 命令而得到的输出结果。

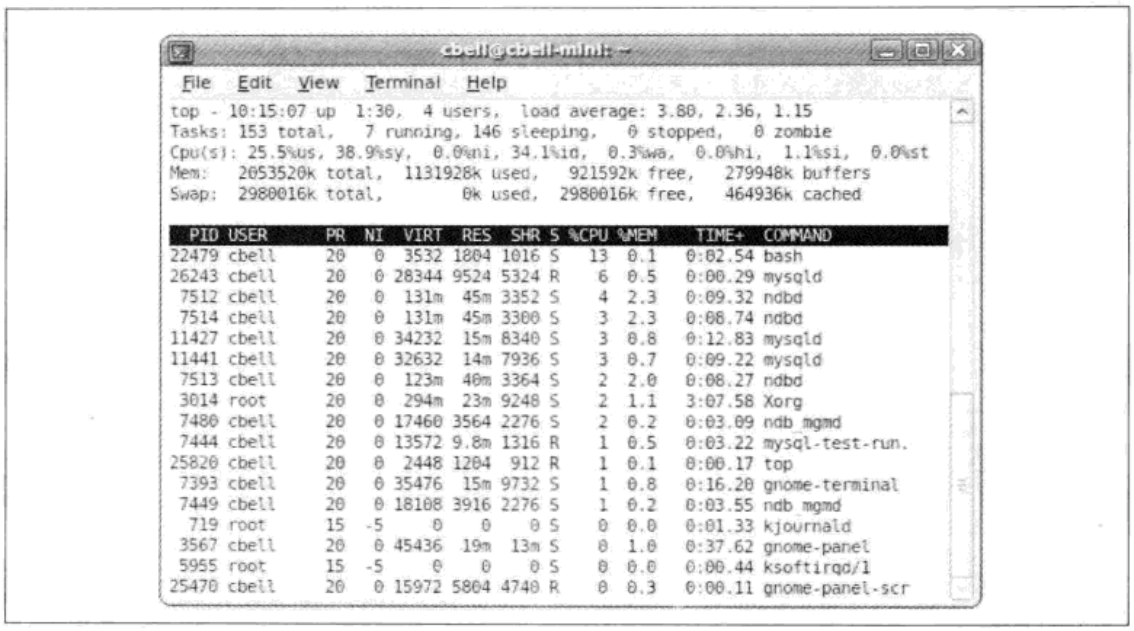


图7-1：top命令

系统概要信息位于列表的顶部，其中含有一些有趣的数据。它分别显示了用户（%us）、系统（%sy）、nice（%ni，这是改变用户进程的优先级所花费的时间）和 I/O 等待（%wa）的 CPU 时间占用百分比，甚至包含了处理软件和硬件中断时占用 CPU 时间的百分比。还包含了内存大小、被占用的内存的大小、可用内存的大小、缓冲区的大小，以及可用交换空间的大小、被占用的交换空间的大小、可用交换空间的大小、交换空间的缓冲区的大小。

系统概述下面是进程列表，这些进程是依据 CPU 占用时间的多少降序排列的（源于这个命令的名称）。在这个例子中，一个 Bash shell 后面正紧跟着一个或几个 MySQL 安装进程。

255

进程优先级（Niceness）

可以改变Linux或者UNIX系统上的进程的优先级。你可能想降低消耗过多CPU的、紧急性低的或可能长时间运行而你又不想取消或重新安排的进程的优先级。这时可以使用nice、ionice和renice命令改变进程的优先级。

如今大部分Linux和UNIX将进程分组，那些改变了优先级的进程被分到nice组。这使你可以获得有关修改了的进程的统计信息，而无须自己记住或整理这些信息。这些报告nice进程的CPU占用时间的命令使你有机会了解这类进程所消耗的CPU量。例如，这个参数值高可能表明至少有一个进程的优先级过高。

top命令的最佳使用方法也许是让它运行并每隔三秒刷新一次。如果你随着时间的推移间歇性地查看信息显示，将首先查看哪些进程消耗最多的CPU时间。这可以帮你一眼就能确定是否存在失控进程。



可以通过指定命令刷新的时间间隔来改变命令的刷新率。例如，top -d 3将命令刷新的时间间隔设置为3秒。

Linux和UNIX的大部分变种中都有类似的top命令。有些含有有趣的互动热键，这些热键让你可以切换信息的开或关、排序列表，甚至将显示改变为彩色的。不同的操作系统的热键和交互功能不同，详见特定操作系统手册中的top命令。

iostat命令

iostat命令为你提供系统的不同信息，包括CPU时间的统计、I/O设备、分区和网络文件系统（NFS）。这个命令对监控进程有用，因为它提供了与进程有关的系统如何整体运作的全景图，其中包括进程信息和系统等待I/O的总时间。图7-2显示了在中等负载下的系统中运行iostat命令的一个实例。

256



iostat、mpstat和sar命令可能不会在你的系统上默认安装，但是可以被选择性地安装。例如，它们是Ubuntu发行版中sysstat包的一部分。有关安装和设置的相关信息详见操作系统文档。

图7-2显示了从系统启动时间起CPU使用率的百分比。这个值是由计算所有处理器的平均值得来的。如你所见，这个系统运行在双核CPU上，却只给出了一行值。这些数据包括CPU利用率的百分比：

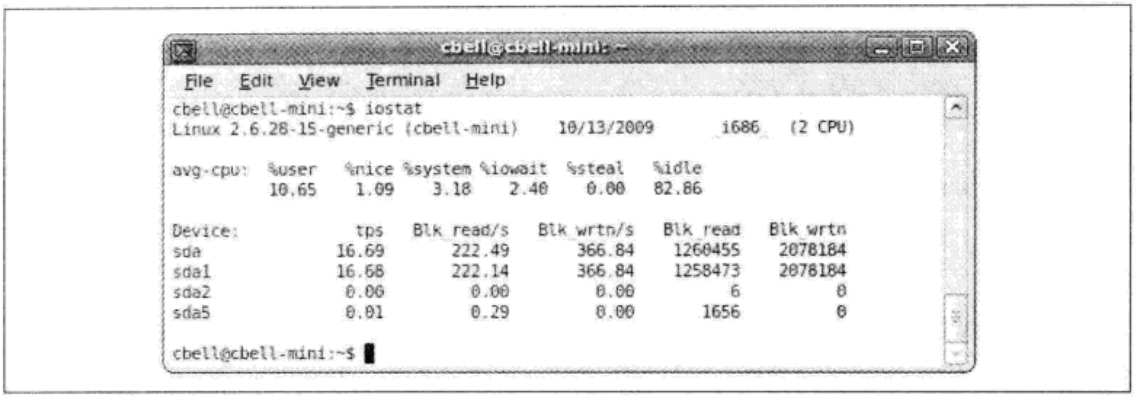


图7-2：isotat命令

- 在用户级别执行（运行应用程序）；
- 以 nice 优先方式在用户级别执行；
- 在系统级别执行（内核进程）；
- 等待 I/O；
- 等待虚拟进程；
- 空闲时间。

类似于这样的报告可以让你了解自启动以来系统是怎样执行的。虽然这意味着你可能无法注意到系统性能差的时段（因为它们一直是平均值），但是在进程怎样消耗可用的处理时间或等待 I/O 这两个方面，它确实提供了独特的视角。例如，如果 `%idle` 低，你可以确定系统一直很忙碌。同样，高 `%iowait` 表明磁盘出现问题。如果 `%system` 或 `%nice` 比 `%user` 高，则表明系统失衡且高级别的进程阻止普通进程的运行。

mpstat命令

257

`mpstat` 命令与 `iostat` 命令显示的处理器时间的信息类似，但是 `mpstat` 将各个处理器的信息分开显示。如果在多处理器系统上运行该命令，将看到所有处理器的总数百分比和单个处理器的数据百分比。图 7-3 显示了 `mpstat` 命令的例子。

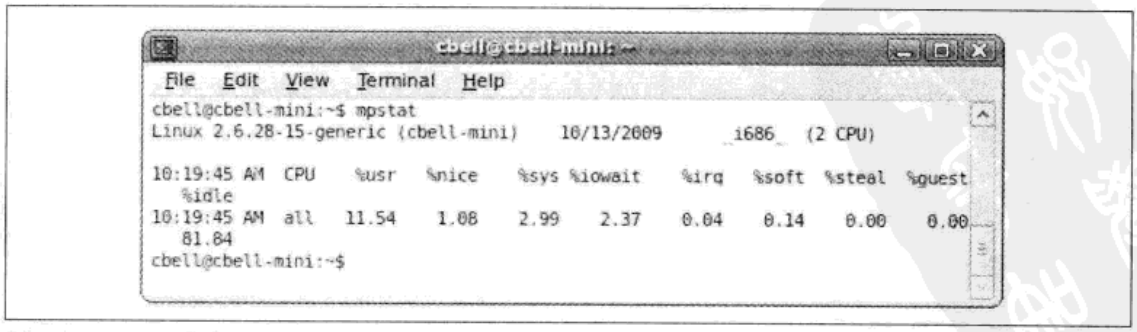


图7-3：mpstat命令

mpstat 命令有个选项可以基于间隔时间刷新信息。这个命令有助于查看处理器在一段时间内是如何执行进程的。例如，使用 mpstat 命令可以查看处理器关联是否失衡（一个特定的处理器被分配过多的进程）。

要了解有关 mpstat 命令的更多知识，请参看操作系统指南。

ps命令

ps 命令是我们日常使用的命令之一，但我们从来都没有花时间去考虑它的能力和效用。这个命令提供系统上正在运行的进程的快照。它显示了进程 ID、进程运行的终端、进程已经运行的时间和进程启动的命令。

ps 命令之所以如此万能，是因其有大量可用于显示数据的选项。你可以显示特定用户的进程，也可以通过显示进程树得到某个特定处理器的相关进程，甚至可以改变其显示结果的格式。请参看你的文档，以获得有关操作系统的 ps 命令可用选项的信息。

使用输出结果来诊断问题的方法之一是寻找长时间运行进程或检查进程的状态（例如，检查那些处于可疑状态或休眠状态的进程）。除非像 MySQL 这样的已知程序，否则你可能要调查它们为什么运行了这么久。

258 图 7-4 显示了在中等负荷的系统中运行 ps 命令的一个简短的例子。

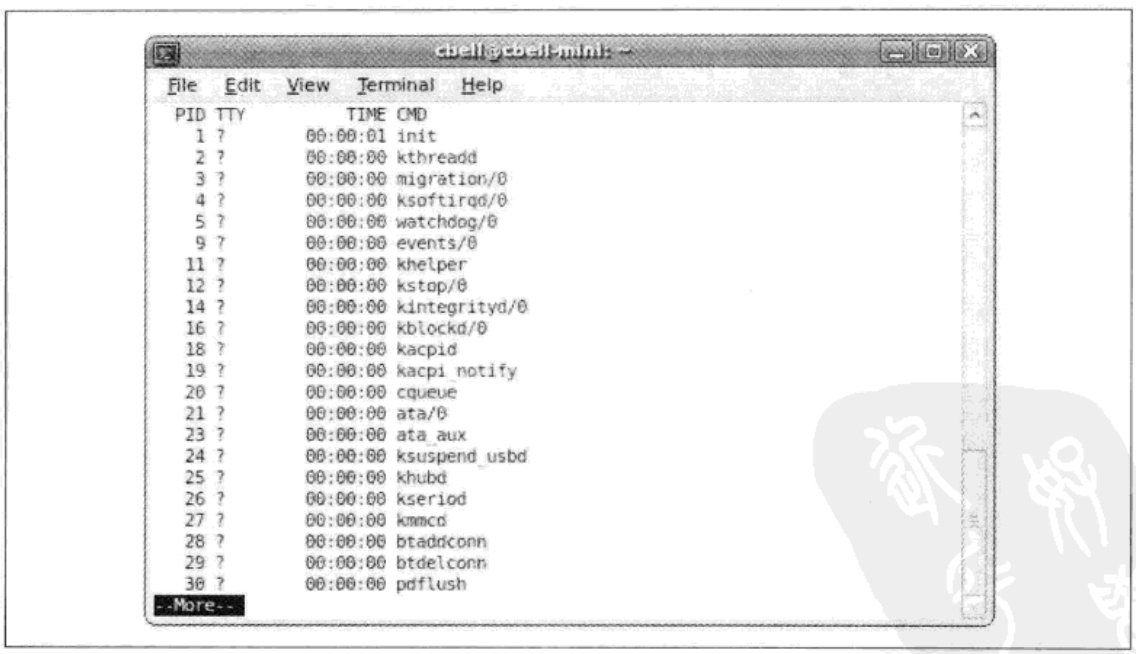


图7-4：ps命令

这些输出结果的另一个用途是：通过它可以了解是否存在一些未知进程或者单个用户是

否运行了很多进程。很多情况下，这表明有一个脚本正在大量产生进程，也许是因为这个脚本是以不合理的方式建立的，甚至可能表明系统存在危险。

也许使用 ps 命令的最常用方式是确定一个特定程序的进程 ID。例如，如果你想知道所有 mysqld 程序的进程 ID，请使用以下命令：

```
ps -A | grep mysqld
```

这将所有的进程列表发送给 grep 命令，grep 命令过滤进程列表，并只显示其中含有“mysqld”的行。可以通过这种方法找到进程的 ID，然后通过其他的命令获取进程的详细信息。

操作体统中还有许多其他内置工具用来显示进程信息。和往常一样，监控进程的更深入信息的最好来源是借鉴特定操作系统的性能调优。

内存利用率

有多个命令提供有关系统内存利用率的相关信息。最流行的是 free 和 pmap。

free命令

free 命令显示可用的物理内存量，其中包括总物理内存量、已用物理内存量、可用物理内存量。它也为交换空间显示同样的统计信息，还显示内核使用的内存缓存大小和缓冲区的大小。图 7-5 显示了在中等负荷的操作系统上运行 free 命令的一个例子。

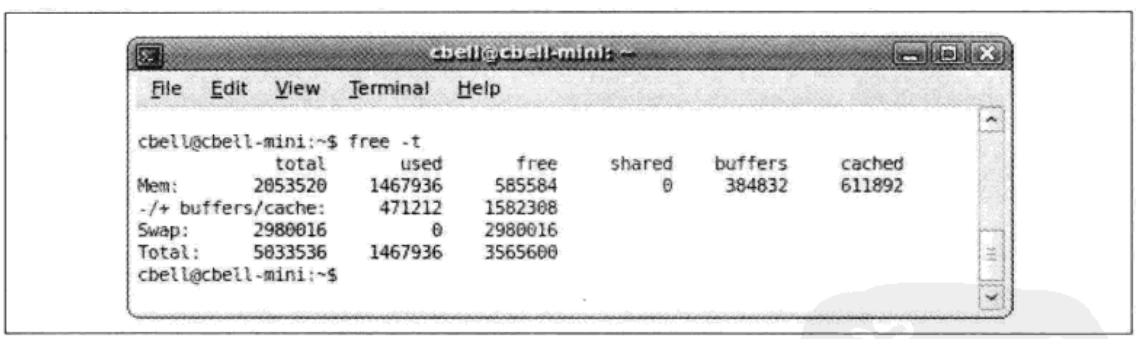


图7-5：free命令



图 7-5 是来自于 Ubuntu 系统的 free 命令的输出结果，其中的 shared 列已经废弃了。

switch 选项将命令设置成轮询模式，使统计信息根据提供的时间间隔秒数定期进行更新。

例如，每隔 5 秒轮询内存一次的命令是 `free -t -s 5`。

pmap命令

pmap 命令提供一个进程所使用的内存的详细映射。要使用此命令，必须先找到你想研究的进程的 ID。你可以通过 ps 命令得到进程的 ID，或者，如果寻找消耗大量 CPU 时间的进程，甚至可以通过 top 命令获得进程的 ID。

也可以通过在命令行中列出进程 ID 列表来获得多个进程的内存映射。例如，`pmap 12578 12579` 命令将会显示进程 ID 为 12578 和 12579 的进程的内存映射。

260 pmap 命令的输出结果显示了所有内存地址的详细信息，且在报告产生的瞬间显示进程使用的内存的大小。它还显示了启动进程的命令，其中包括完整的路径和参数，可以用于确定进程是从什么地方开始的和进程使用了哪些选项。当你试图去弄清楚进程为什么行为异常时，会惊奇地发现这个命令是多么方便。该显示也展示内存块的模式（访问权限），这在诊断进程间的问题时非常有用。图 7-6 和图 7-7 显示了在中等负荷的系统上运行 mysqld 进程的一个例子。

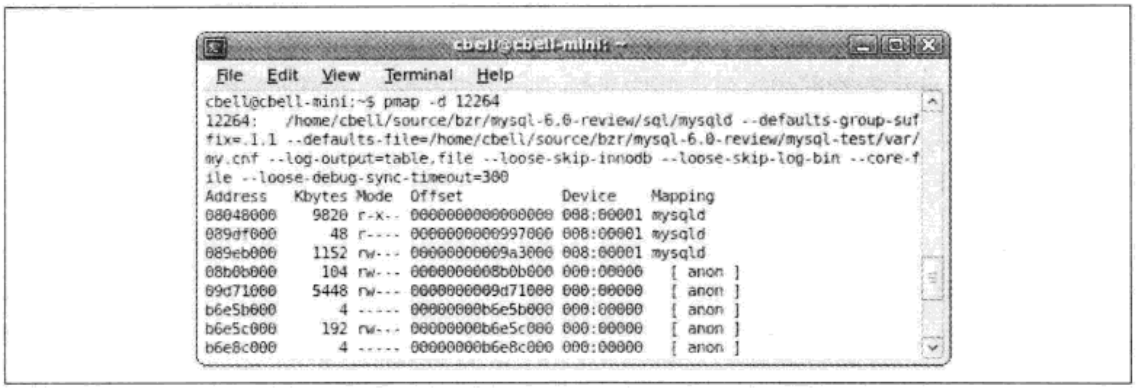


图7-6：pmap命令——第一部分

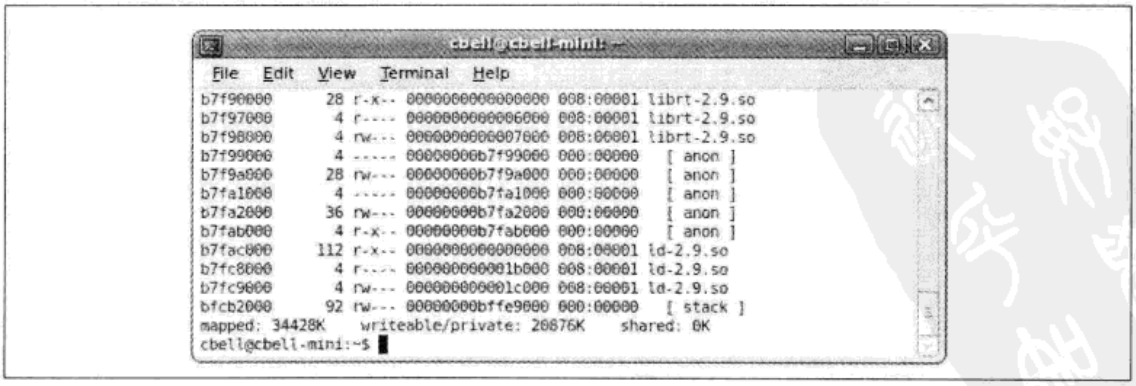


图7-7：pmap命令——第二部分

请注意，图中的列表显示了设备输出格式（通过在启动时添加 -d 参数来选择），也显示了内存在哪里被映射的或使用的。pmap 命令可以用于诊断一个特殊的进程为什么消耗非常多的内存和诊断哪个部分（例如一个库）消耗内存最多。

图 7-7 显示了 pmap 命令输出的最后一行，这显示了一些有用的概要信息。 261

最后一行显示了有多少内存被映射到文件、私有内存空间量和与其他进程共享的内存量。这些信息也许是解决内存分配和共享问题的关键数据。

还有一些其他显示内存利用率的命令和实用工具（例如 dmesh，可以在开机启动时显示信息），请参看有关你的操作系统的性能调优方面的资料。

磁盘利用率

一些命令可以显示你的系统上的磁盘利用率的统计信息。本节将描述并演示 iostat 和 sar 命令。

iostat命令

正如已经在前面的“进程活动”中描述的一样，iostat 命令显示被占用的 CPU 时间、所有磁盘的列表，以及它们的统计信息。具体来说，iostat 显示设备列表、设备的传输速度、每秒读写块的数量和读写块的总数量。为了便于查找，图 7-8 重复显示图 7-2，是一个在中等负荷的系统上运行 iostat 命令的例子。

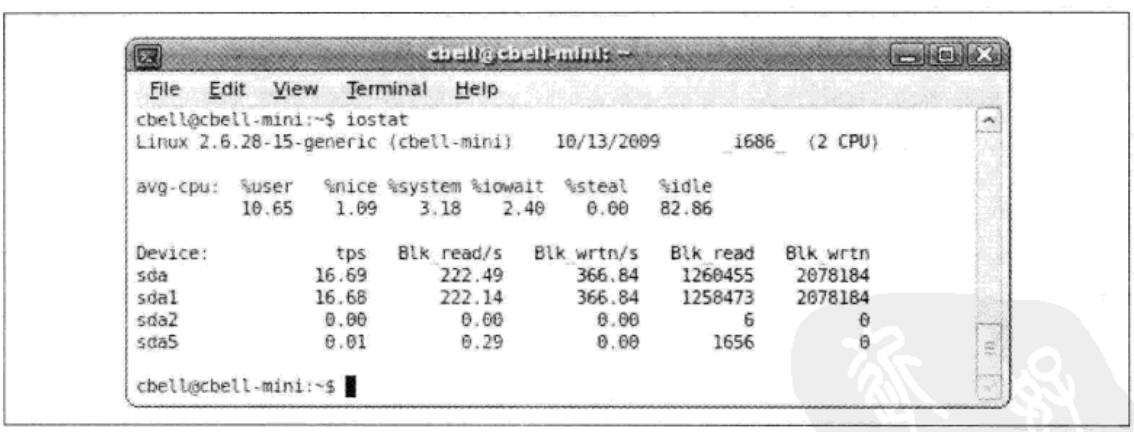


图7-8：iostat命令

在诊断磁盘问题时，这份报告是非常重要的。只需看一下，它就可以告诉你是否有些设备比其他的使用得要多。如果是这种情况，可以将一些进程移到其他的设备上，从而减少对单个磁盘的请求量。这个输出结果也可以告诉你哪个磁盘的读写量最多，帮助你确

定是否有特定的设备需要升级为更快的设备。相反，也可以了解到哪个设备没有被充分利用。例如，如果发现闪亮的新的超快速磁盘没有被频繁访问，那么有可能是你还没有将高容量的进程配置到新磁盘上。也有可能是你的程序使用了缓存，以至于很少执行 I/O。

sar命令

sar 命令是一个非常强大的工具，能够显示有关系统的各种信息。由于它记录了一段时间的数据，且能够以各种不同的方式进行配置，所以安装它可能有点棘手。请参考操作系统文档，以确保正确地安装它。像我们描述的大部分系统命令一样，也可以配置 sar，让其定期生成报告。



sar 命令也可以显示 CPU 利用率、内存、缓存和其他一些主机信息，就像其他命令显示的信息一样。一些管理员将 sar 设置为定期运行的模式，并提取数据以形成系统的基准。有关 sar 的完整指南已超出了本书的讨论范围。如需要更详细的研究，请参阅由 Gian-Paolo D、Musumeci 和 Mike Loukides 编写的“*System Performance Tuning*”(O'Reilly, *http://oreilly.com/catalog/9780596002848/*)。

本节将看如何使用 sar 命令来显示磁盘使用率的信息。为此，还结合显示 I/O 传输率、交换空间和分页的统计信息，以及块设备的使用率。图 7-9 显示了使用 sar 命令显示磁盘使用情况统计的一个例子。

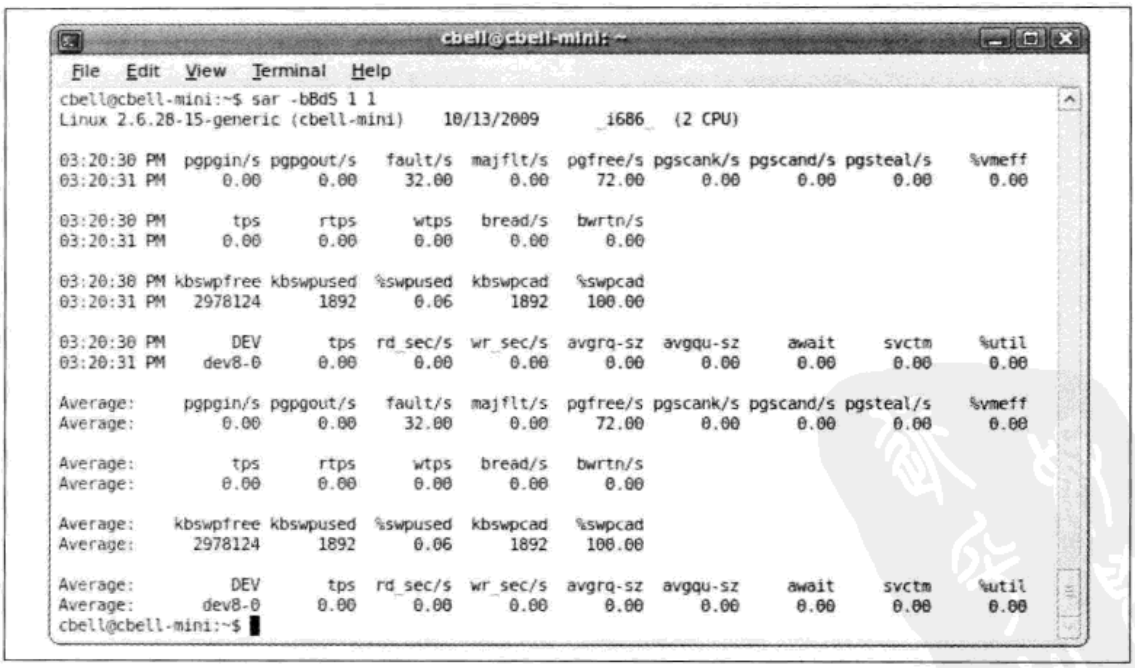


图7-9：有关磁盘使用率的sar命令

这个报告显示如此多的信息，以至于乍一看，它似乎很巨大。请注意头信息后的第一部分，这是有关分页信息的报告，首先显示了分页子系统的性能，下面是关于 I/O 传输率的报告，紧接着是关于交换空间的报告，然后是有关设备及其统计信息的列表。这个报告的最后显示了所有参数样本的计算平均值。

有关分页的报告显示了翻出或翻入内存的页面率、每秒内不需要磁盘访问的分页的错误数、需要磁盘访问的重大故障数和有关分页系统性能的其他统计信息。这些信息很有用，当发现大量分页错误时（重大分页错误代价更大），很可能表明系统运行了过多的进程。大量的严重的分页错误可以导致磁盘使用问题。也就是说，如果这个值很高且磁盘使用率很高，这有可能不是磁盘性能差导致的，而仅仅是应用程序或操作系统中某些地方出错而导致的。

I/O 传输率报告显示了每秒的事务数量、读写请求和读写块的总数量。在这个例子中，系统没有使用 I/O 而处在高 CPU 负载下。这表明系统健康。如果 I/O 值很高，系统中有可能存在一个或多个进程被卡在了 I/O 绑定状态。MySQL 上的查询产生的大量随机磁盘访问或横跨多个磁盘碎片的表都可能会导致以上问题。 263

交换空间的报告显示了可用交换空间的大小、被使用的交换空间的大小及使用百分比和内存使用量。它有助于识别交换出过多进程的问题，另外，就像其他报告显示的那样，它还可以帮助确定问题是由磁盘和其他设备导致的，还是由内存或过多的进程导致的。

块设备（系统中以块的方式移动数据的区域，如磁盘、内存等）的报告显示了传输速率（tps）、每秒的读写速率和平均等待时间。这些信息有助于诊断系统块设备问题。如果这些值都很高（不像这个例子显示的那样几乎没有任何设备活动），可能意味着系统已经达到了设备的最大带宽。然而，这些信息必须结合报告上其他信息才能判断系统是否超负荷——运行太多进程或没有足够内存（或者两个问题都同时存在）。

该综合报告可以帮助你确定磁盘使用问题出在哪儿。如果分页报告显示错误率异常高，表明系统可能运行了太多的应用程序或者没有足够的内存。但是，如果这些值较低或一般，你需要查看交换空间信息，如果交换空间的值也正常，可以查看设备使用报告的异常信息。

磁盘使用分析器 264

除了操作系统实用程序以外，GNOME 桌面项目还创造了一个图形应用程序——磁盘使用分析器。这个工具让你深入了解存储设备是如何被使用的，还展示了描述磁盘使用情况的图形。这个实用工具在大多数 Linux 发行版本中都可用。

图 7-10 显示了磁盘使用分析器的一个报告样本。

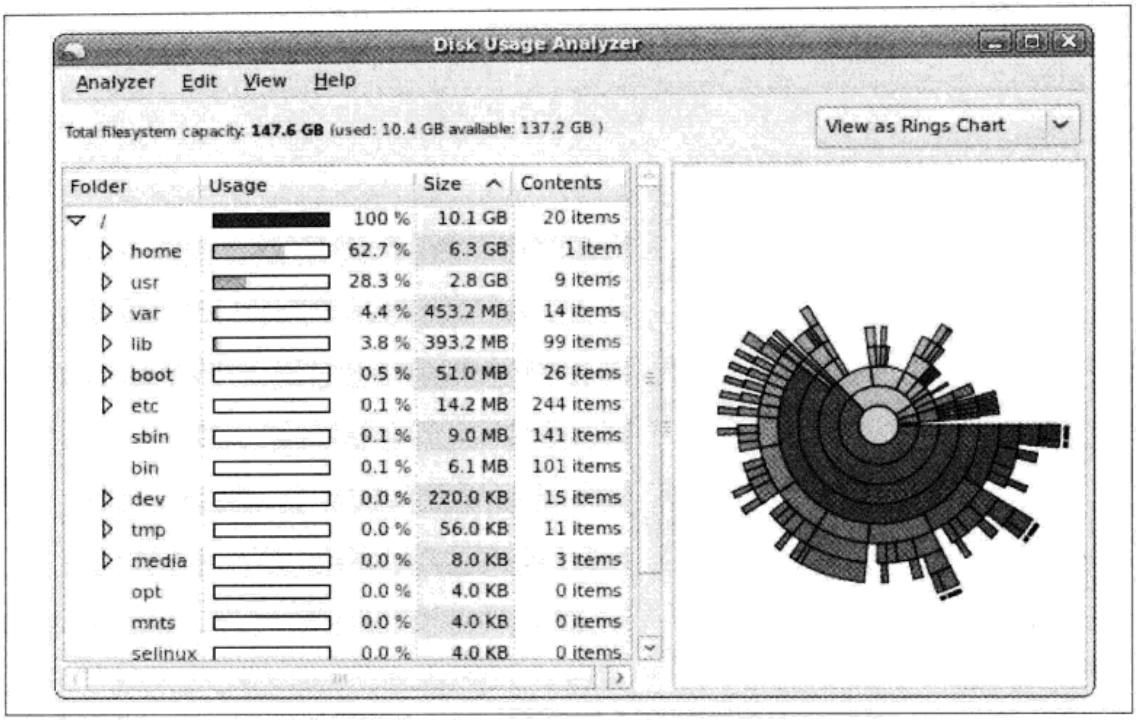


图7-10：磁盘使用分析器

基本上这个报告显示了设备是如何依靠分页和交换系统而运行的。当然，如果一个系统交换过多的进程进出内存，磁盘使用情况将会出现异常。这就是为什么在同一张报告上看到这些是有价值的。

诊断磁盘问题是具有挑战性的，只有几个命令显示磁盘使用情况的详细统计资料。然而，有些操作系统提供了更详细、更专业的磁盘使用情况检测工具。不要忘记，也可以通过许多常见命令（如 `ls`、`df` 和 `fdisk`）来确定可用空间大小、什么被挂载、每个磁盘有哪些文件系统及更多信息。参看你的操作系统文档，可以了解所有与磁盘相关的命令的列表和描述，以及磁盘使用率和监控命令。

265



本章后面所示的 `vmstat` 命令，也可以显示这些数据。使用 `vmstat -d` 命令可以得到这些数据的文本。

网络活动

网络活动问题的诊断可能需要专业的硬件和网络协议知识。详细的诊断通常留给网络专家，但是作为 MySQL 管理员，可以使用 `netstat` 和 `ifconfig` 这两个命令获取网络的基本信息。

netstat命令

netstat命令可以显示网络连接、路由器、接口统计数据和其他网络相关的信息。netstat命令提供了很多的信息，网络专家将这些信息用于诊断和配置复杂的网络问题。不过，它可以帮助你了解有多少流量正在通过网络接口和哪些接口被访问得最多。图7-11显示了所有网络接口的样本报告并显示了每个接口的数据传输量。

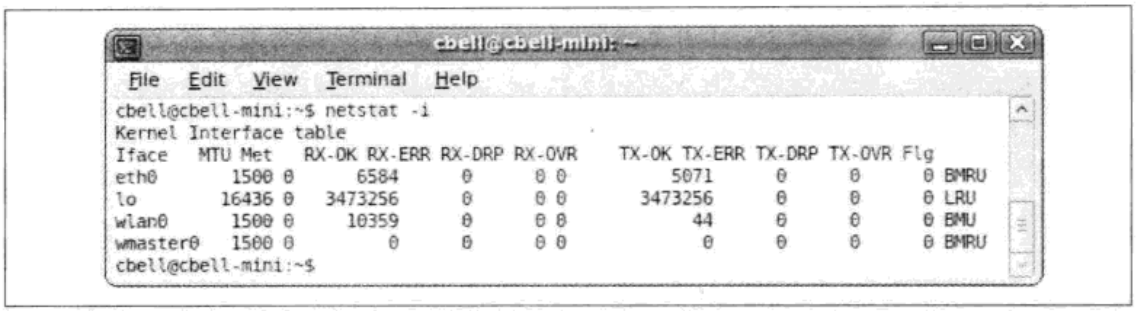


图7-11：netstat命令

系统中有多个网络接口，netstat命令显示的信息有助于确定某个接口是否被过度使用或者是否有错误的接口被激活。

ifconfig 命令

ifconfig命令是任何网络诊断必不可少的工具，它显示系统中网络接口的列表，其中包括每个网络接口的状态和设置。图7-12显示了ifconfig命令的一个例子。

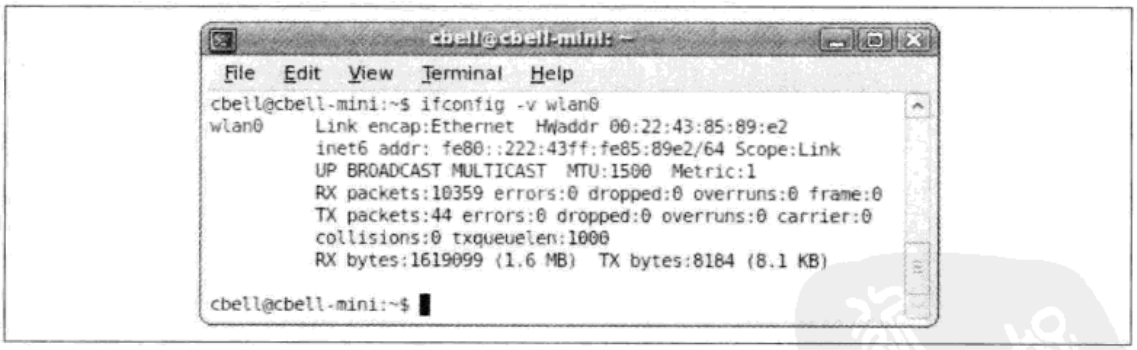


图7-12：ifconfig命令

请注意每个接口，无论是被激活或被禁止的都被列出来了。这些信息是非常有用的，可用于确定一个接口是怎样被配置的，例如，它可以告诉你网络已经故障转移到一个更慢的接口，而不是通过超高速以太网适配器进行通信。很多时候，网络问题的根源不是网络流量，而是网络接口的选择和配置。

266

如果在需要专家帮助诊断网络问题时，提前生成了网络系统报告，这些报告数据将可以帮助网络专家集中精力更快地处理问题。一旦减少了消耗太多网络带宽的进程，并确定有一个可行的网络接口，网络专家就可以通过接口配置来优化网络性能。

常见系统统计信息

我们已经讨论了各个子系统的具体命令，并将这些统计报告命令进行了分组。Linux 和 UNIX 还提供了一些额外命令，其中包括 uptime 和 vmstat，这些命令可以显示更多的常见系统信息。

uptime命令

uptime 命令显示了系统运行了多长时间，还显示了系统当前时间、系统运行了多长时间、有多少用户正在使用（登录）该系统和系统每隔 1 分钟、5 分钟、15 分钟的平均负载。图 7-13 显示了该命令的一个例子。

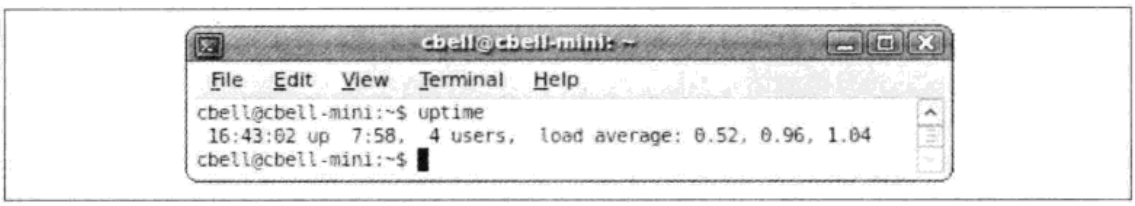


图7-13：uptime命令

267 如果想了解最近一段时间内系统的平均执行能力，可以使用 uptime 命令获取相关信息。信息中给出的平均负载是针对处于活动状态的进程而言的（非等待 I/O 或 CPU 状态）。因此，这个信息用于确定性能问题是有限的，但是可以显示一般意义上的系统健康状况。

vmstat命令

vmstat 命令是一个通用的报表工具，提供有关进程、内存、分页系统、I/O 块和 CPU 活动的信息。它有时被用在确定性能问题的第一步。一些字段的值过高可能会引导你使用本章中讨论的其他命令去做更深入地检查。

图 7-14 显示了在低负荷系统上运行 vmstat 命令的例子。

此处显示的数据包括进程的数量，其中 r 表示那些等待运行的进程，b 表示那些处于不间断状态的进程。下一列显示了交换空间的总量，其中包括了交换出（si）或交换入（so）的内存量。下面的区域显示了块接收（bi）或发送（bo）的 I/O 报告。再下一个区域显示了每秒的中断数（in）、每秒的上下文切换数（cs）、用户空间上进程运行时间（us）、

内核空间上进程运行的时间（sy）、闲置时间（id）和等待 I/O 的时间（wa）。这些时间都是以秒为单位的。

还有更多有关 vmstat 命令的参数和选项，详见操作系统手册。

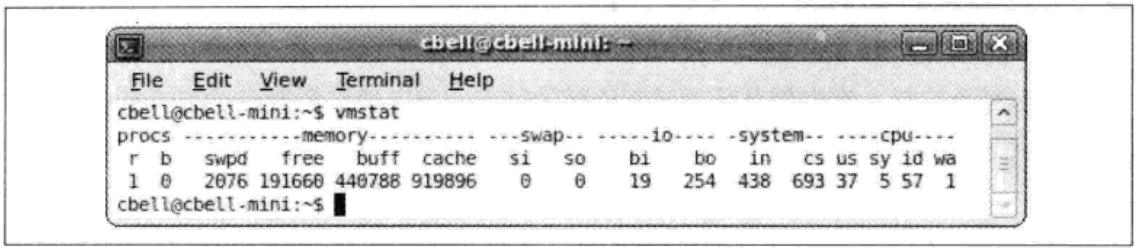


图7-14：vmstat命令

使用cron自动监控

268

也许最需要重点考虑的工具是 cron。如前面的“UNIX 任务调度”那一节描述的一样，可以使用 cron 安排进程在特定时间内运行。这允许你运行命令并为后期的分析保存报告结果。cron 是一个非常强大的工具，可以获得一段时间内的系统快照。然后，可以利用这些数据形成系统参数的平均值，当系统日后出现性能不佳时，就可以用这个均值作为基准，与当前系统参数比较。这很重要，因为它让你一眼就看到哪些发生了变化，并节省大量诊断性能问题的时间。

如果每天都运行性能监控工具并检查其结果，然后将结果与基准进行比较，可能在用户开始抱怨前就可以侦查出系统的问题。事实上，这是我们描述的主动监控的基本前提。

Mac OS X 监控

由于 Mac OS X 操作系统是基于 UNIX Mac 内核开发的，因此可以使用前面介绍的大部分工具监控 Mac OS X 操作系统。然而，有一些其他特别针对于 Mac 而设计的监控工具，包括以下图形管理工具：

- System Profiler
- Console
- Activity Monnitor

本节将概述用于监控 Mac OS X 操作系统的工具。这些工具构成了 Mac OS X 系统的核心监控和报告工具。在良好的 Mac 设计方案中，它们被精心编写，并拥有良好的用户交互界面（GUI）。这些 GUI 甚至显示了从文件中报告信息的部分工具。正如你将看到的，每个工具都非常有用，并且能够帮助你诊断 Mac 系统上的性能问题。

System profile

System profile 提供了系统状态的快照。它提供了有关系统的一切详细信息，其中包括所有的硬件、网络和安装的软件的相关的信息。图 7-15 显示了 System Profile 的一个实例。

可以在磁盘驱动器的实用程序 / 实用工具（*Applications/Utilities*）文件夹中找到 system profile。也可以通过 spotlight 启动 system profile。如图 7-15 所示，该工具的左侧提供了一个树形菜单窗口，而右侧是显示详细资料的窗口。可以通过树形菜单查看系统的各个组件的详情。

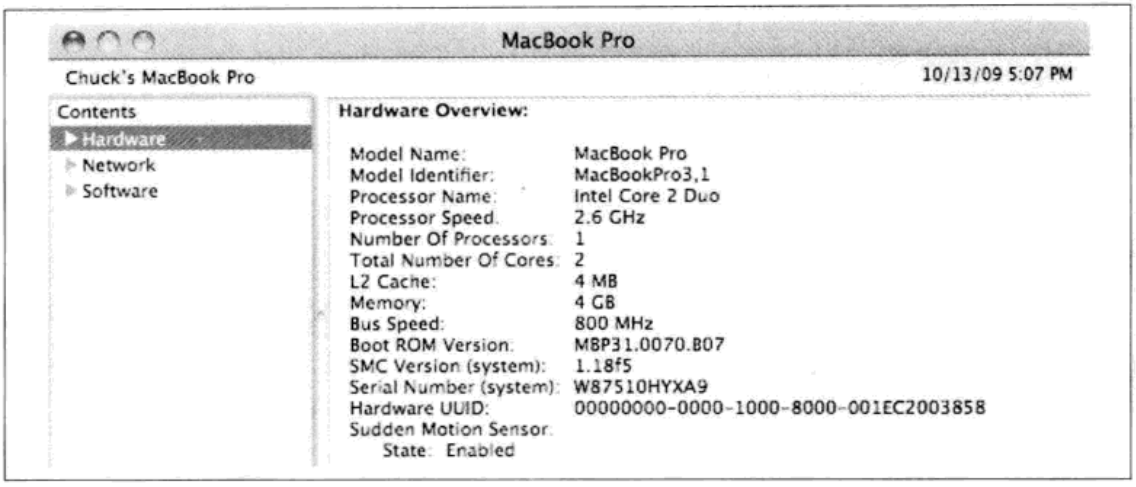


图7-15：system profile

269



如果你更喜欢基于控制台的报告，Mac 提供了一个基于命令行的同等应用程序 */usr/bin/system_profiler*。这里有很多参数和选项，可以用于限制某些报告的显示。要了解更多的信息，请打开终端，然后输入 `man system_profile`。

如果打开硬件树，将会看到系统上的所有硬件的列表。例如，如果想看到安装在系统上的内存的类型，可以单击硬件树上的内存条目。

System Profile 提供了网络报告，我们将在 Linux 上以另外一种形式看到该网络报告信息。单击网络树以获得系统上所有网络接口的基本报告。选择网络树上或者详细信息窗口上的一个网络接口，可以看到该信息与在 Linux 和 UNIX 上通过网络命令产生的信息相同。还可以找到关于防火墙的信息和定义的地点信息，甚至可以找到网络共享卷的信息。

另一个非常有用的报告显示了系统上安装的应用程序。单击 Software → Application report，将看到系统上所有软件的列表，其中包括软件名、版本、更新时间、是否是 64 位应用程序和软件类型（比如，是通用类型还是本地 Intel 二进制类型）。软件类型是非

常重要的信息，例如，普通二进制类型比 Intel 二进制类型运行得慢。提前知道这些是比较好的，因为它们可以为性能设置一定的期望值。

图 7-16 显示了这个报告的一个实例。

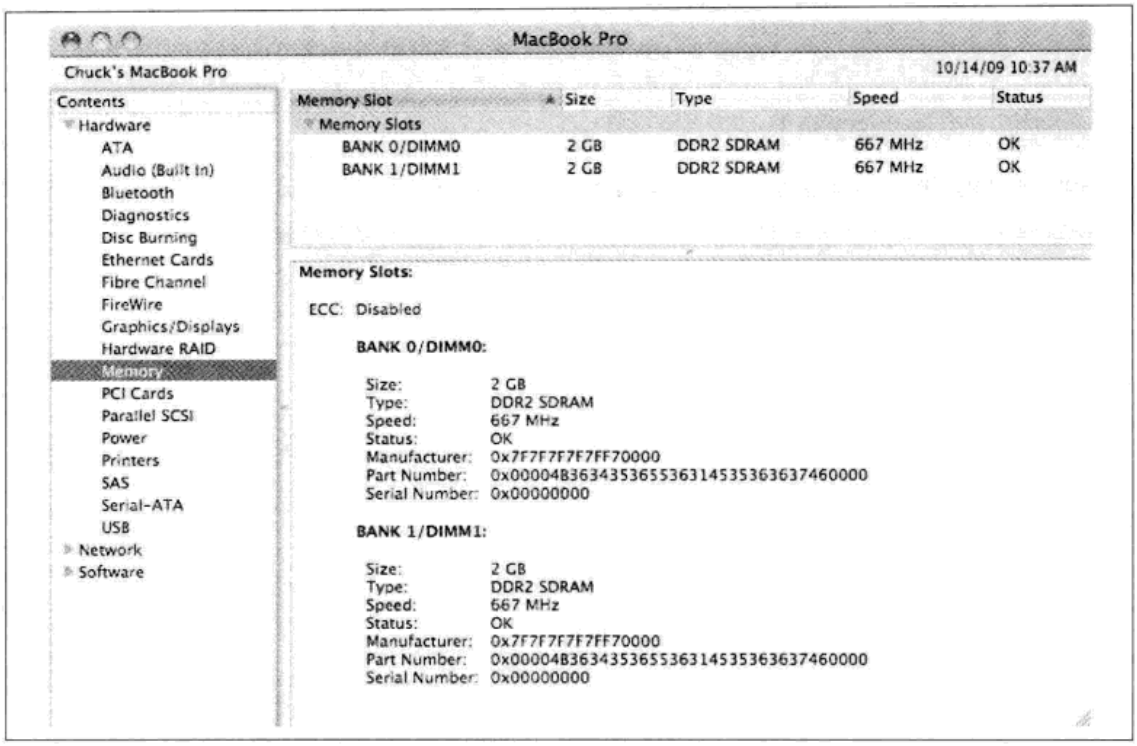


图7-16：来自System Profile的内存报告

如你所见，这里有大量的细节信息。可以看到系统安装了多少内存卡和这些内存卡的速度，甚至可以看到它们的制造商代码和部件编号。 270



我们称每个细节窗口为一个报告，因为它本质上是一个给定类别的详细报告。有些人可能将所有的数据看成一个报告，这样做是不明智的，我们认为将所有的东西看成报告的集合会更好。

如果你对这个工具的功能感兴趣，请在这个树上任意实验和挖掘以获取更多有关系统的信息。你将在这里发现几乎所有的事实。

在诊断系统问题时，System Profile 是非常有价值的。很多时候，AppleCare 代理人和训练有素的 Apple 技术人员将索要你的系统报告。通过使用 File → Save 命令从 System Profile 产生这些报告，并将报告保存为 Apple 专业人士可以使用的 XML 文件。也可以通过使用 File → Export 命令将报告输出为 RTF，并且可以把报告保存为 PDF 后打印。

271 还可以通过使用 View 菜单更改详细报告的级别。它有 *mini*、*basic* 和 *full* 这些级别，通过这些选项可以将报告的详细级别从最小级别提升到一个完整的报告级别。Apple 专业人士通常会索要完整的报告。

System Profile 报告是不需要拆开机箱而能确定你的系统上有什么的最佳方法。它应该是你确认系统配置的第一个来源。

控制台（Console）

控制台应用程序显示了系统上的日志文件信息，位于 */Appliction/Utilities* 文件夹或 spotlight 下。与 System Profile 不同，这个工具不仅提供数据转储，而且能够搜索重要日志信息。它有助于诊断问题，可以通过它查看是否日志中有更多有关事件的信息。图 7-17 显示了 Console 应用的一个实例。

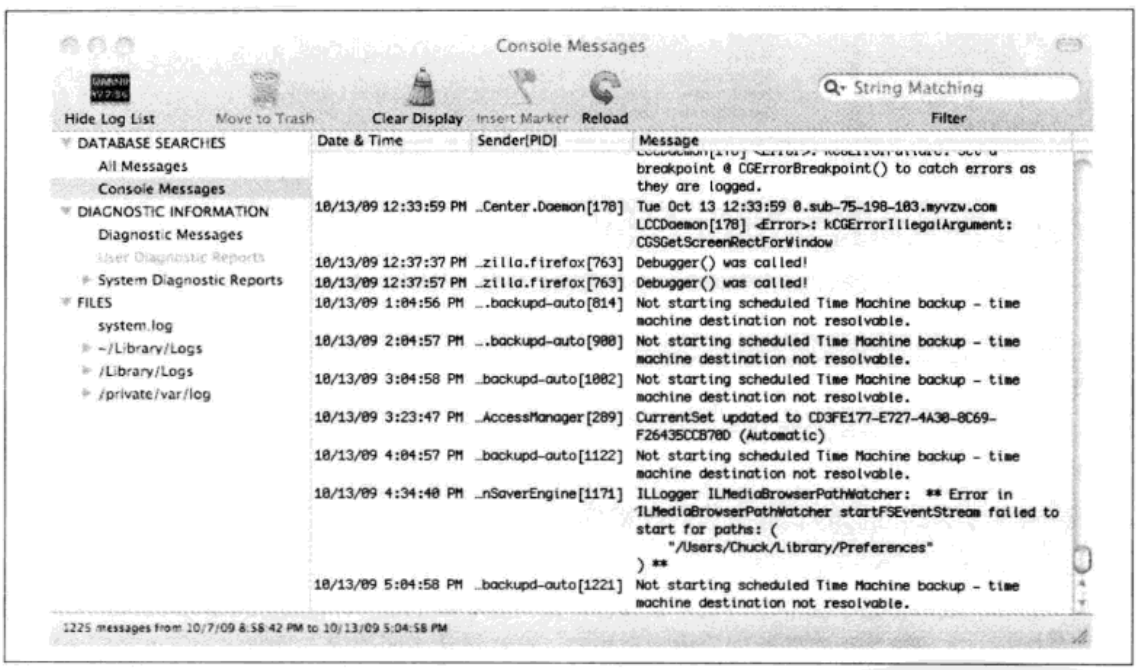


图7-17：Console应用

当启动 Console 应用程序时，它将读取所有的系统日志，并将系统日志归类到 console 诊断消息内。如图 7-17 所示，左边显示日志搜索功能，右边显示日志视图。也可以单击文件树上的 individual logfiles，然后查看每个日志的内容。日志文件包括以下内容：

- *~/Library/Logs*
 存储所有与用户应用程序相关的消息。其中包括登录时崩溃的应用程序、磁盘活动信息，以及其他与用户相关的任务。

- */Library/Logs*
 存储所有系统信息，这里可以查看系统崩溃级别产生的信息和其他异常事件相关的信息
- */private/var/log*
 存储 UNIX BSD 进程相关的信息。在此可以查看系统守护进程或 BSD 实用工具的相关信息。



日志是个连续的文本，不断被追加新的数据，从来不会从中间更新，且很少被删除。

Console 最强大的功能是它的搜索能力。可以创建包含给定短语或关键字的信息的报告，以便日后查看。要创建一个新的搜索，可以选择菜单中的 File → New Database Search，然后将看到一个广义的搜索生成器，可以用它来创建查询。当完成查询后，可以命名并保存报告以便以后处理，这是监控棘手的应用程序的一个非常方便的方法。

Console 另一个很好的特性是在日志中可以标记显示当前时间和日期的标志，可以以此确定最后一次查看日志的时间。你可能和我们有一样的经历，经常会在日志的某些地方发现有趣的信息，需要日后回顾它们，但是却不知道在哪里找到它们，或者不知道日志看到哪里了，在这种情况下，有标记日志的习惯是很有帮助的。标记日志时，先选中文件中需要标记的地方，然后按一下工具栏中的标记按钮。

272

虽然报告的数据是日志启动后的静态快照，而且你运行的任何报告都局限于此快照，但是也可以为日志中的新信息设置警报。请通过 Console → Preferences 打开通知，这些通知在一段时间后通过 Dock 的弹跳图标或 Console 应用程序传送到前台。

Console 应用程序是非常有用的，通过监控发生的事件可以查看系统的各个方面，还可以利用 console 查找应用程序或硬件的错误。当你面对一个性能问题或其他棘手的事件时，一定要搜索关于应用程序或事件的日志信息。有时候，问题的解决方法就包含在摆在你面前的应用程序产生的消息中。

Activity Monitor

273

Activity Monitor 不像先前描述的工具的静态特性那样，它是一个动态工具，能够提供正在运行的系统的相关信息。可以在 Activity Monitor 中找到解决性能问题所需要的大量数据。事实上，当仔细查看 Activity Monitor 时，会看到与在 Linux 和 UNIX 章节提到的所有工具显示的内容相类似的信息，包括 CPU、系统内存、磁盘活动、磁盘使用率和网络接口。

例如，通过 Activity Monitor 可以找到哪些进程正在运行，以及每个进程消耗的内存和 CPU 时间百分比。在这种情况下，它的作用与 Linux 的 top 命令类似。

CPU 面板显示了一些有用的信息，如在用户空间（用户时间）执行所花费的时间的百分比、占用系统空间（系统时间）的百分比和空闲时间的百分比。该展示图还显示了线程数和运行中的进程数，还有一张显示用户和系统的时间总和的彩色编码图。它与 top-like 面板相结合有助于调查与 CPU-bound 进程相关的问题。

图 7-18 显示了 Activity Monitor 显示的 CPU 报告。

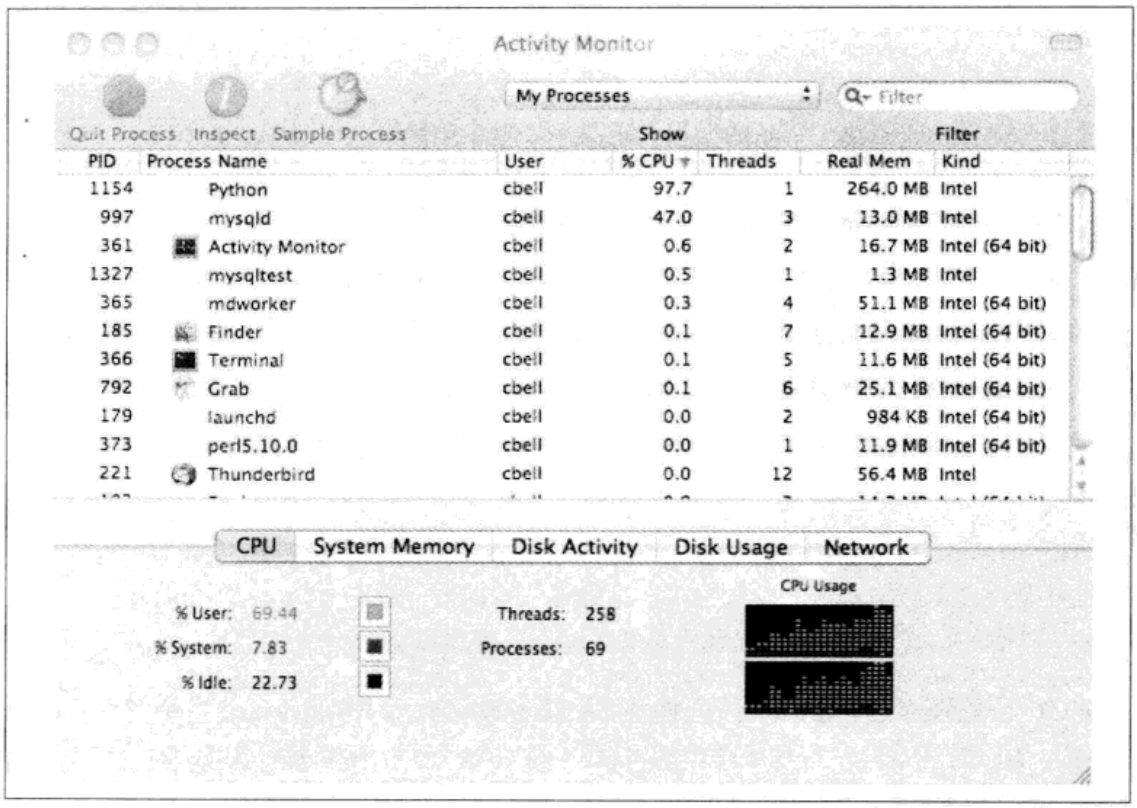


图7-18：Activity Monitor的CPU展示

274 注意，这里有个 Python 脚本，在采样的时间内消耗了相当大的 CPU 时间，在这种情况下，系统在终端窗口运行了一个 Bazaar 分支。Activity Monitor 显示了系统为什么在展开代码树时变得缓慢。

可以双击进程以获得更多有关该进程的信息。也可以通过控制方式或强制退出的方式取消进程。图 7-19 显示进程监测对话框的实例。

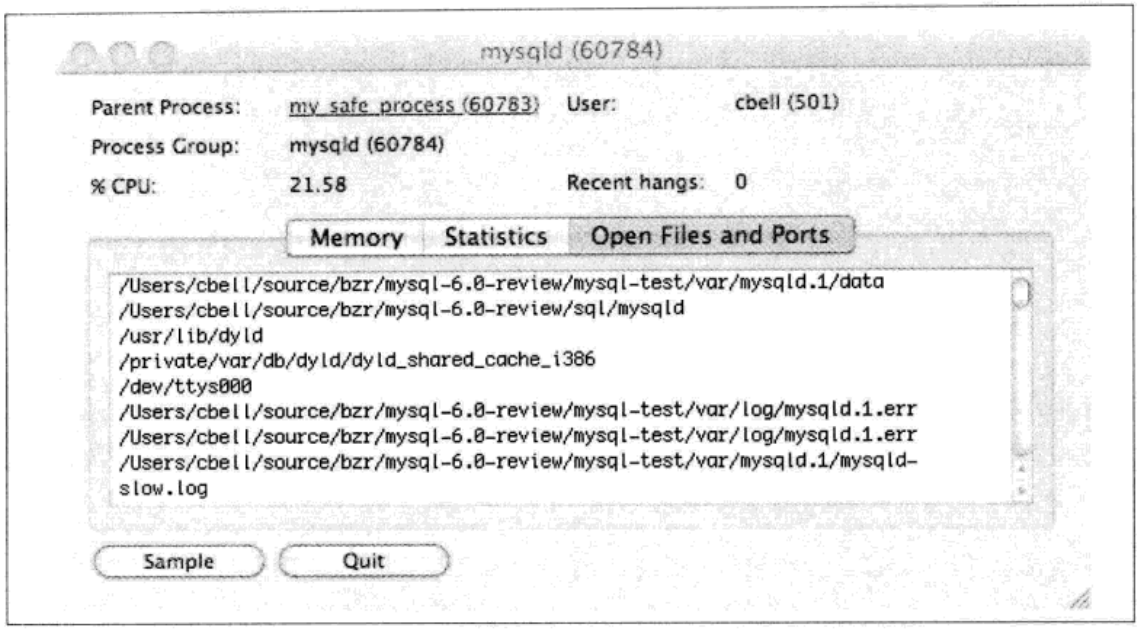


图7-19：Activity Monotor的进程监测对话框



可以通过选择 File → Save 输出进程列表。可以以文本或 XML 文件形式保存进程列表。在诊断问题时，一些 Apple 专业人士可能除了需要 System profile 报告外，还需要进程列表文件。

系统内存面板（如图 7-20 所示）显示内存分布的信息。它显示了多少内存是空闲的、多少内存不能被缓存而必须留在 RAM（换句话说，联动内存）、多少内存被占用，以及多少内存是无效的。通过这个报告，可以一眼看出系统是否存在内存问题。

磁盘活动面板（如图 7-21 所示）显示了所有磁盘的磁盘活动。在第一列中显示了读取磁盘数据的总传输量，以及磁盘每秒的读写性能。下一列显示了读写磁盘数据的总大小，以及每个磁盘的吞吐量，还显示了一段时间内读取数据的彩色编码图。

275

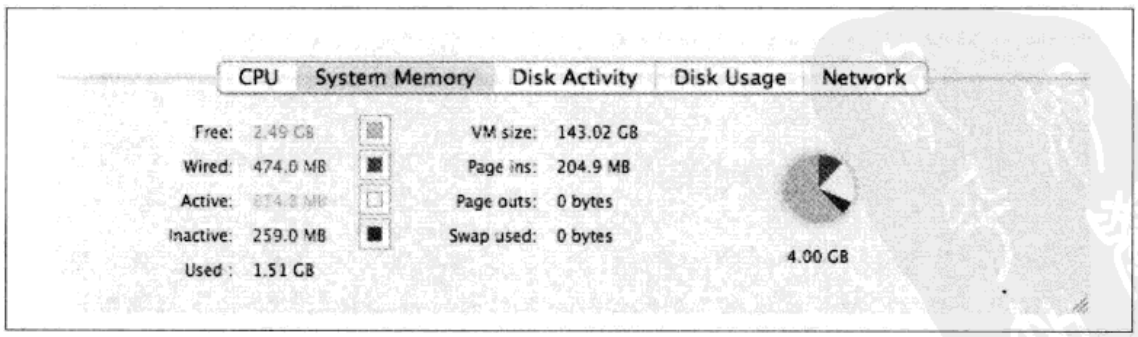


图7-20：Activity Monitor的系统内存面板

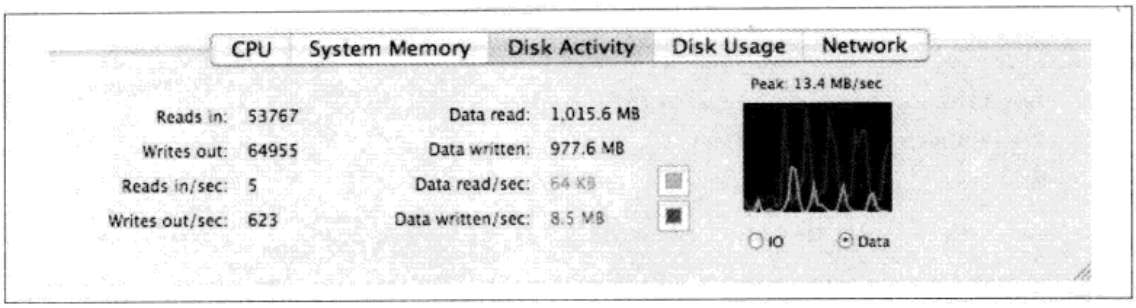


图7-21：Activity Monitor的磁盘活动面板

磁盘活动数据可以显示系统是否调用了很多磁盘访问，以及系统读写量（数据总量）是否异常地高。一个异常高的值可能表示你需要在不同时间运行进程以使它们不去争用磁盘，或者表示需要添加其他磁盘以使系统达到负载平衡。

磁盘使用率面板（如图 7-22 所示）显示了每个驱动器的可用和已用空间大小，还显示了一个彩色的磁盘使用率的饼状图，以方便你快速查看磁盘的使用率。可以通过选择下拉列表中的磁盘查看磁盘的使用率。

这个面板允许监控磁盘上的可用空间，以至于当系统运行缓慢时，让你知道何时需要添加更多的磁盘或何时扩展分区以增加更多的空间。

网络面板（如图 7-23 所示）显示了有关系统怎样与网络进行通信的大量信息。第一列显示了通过网络接收（读）或发送（写）的数据包量。还显示了每秒读写数据包的性能统计。下一列显示了网络上读写数据的大小，以及各个方向的传输速率。另外还有一张彩图显
276 示了网络的相对性能。请注意图中的峰值，可以使用这些数据确定是否有进程消耗了系统网络接口的最大带宽。

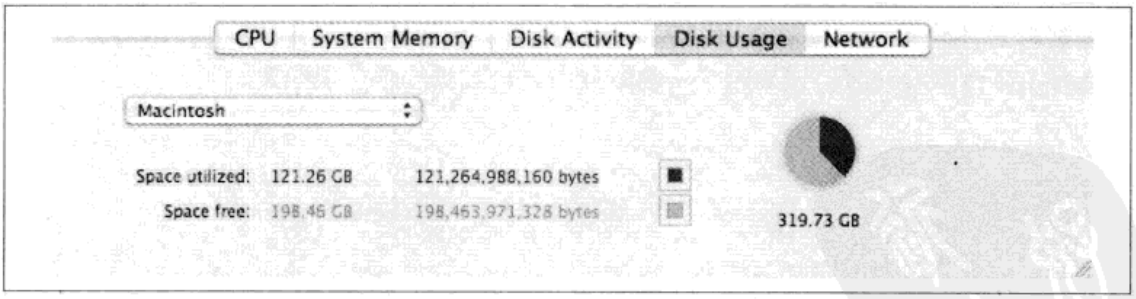


图7-22：Activity Monitor的磁盘使用率面板

本节简单介绍了 Mac OS X 上的强大监控工具。虽然不是一个完整的指南，但是它有助于监控 Mac OS X。要了解每个应用的完整细节信息，必须参看由 Apple 上每个应用程序的 Help 菜单提供的文档。

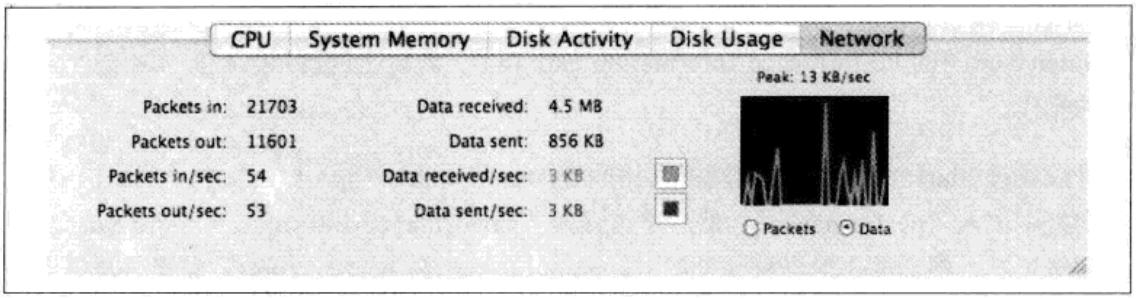


图7-23：Activity Monitor的网络面板

Microsoft Windows监控

Windows 背负着缺乏工具的名声，有人称其监控是违反直觉的。事实上，Windows 附带了一些功能强大的工具，其中包括一个运行任务的调度工具。使用这些工具可以进行性能快照、在事件查看器中检查错误（相当于 Windows 的日志），以及实时监控系统性能。



本节中显示的图片是从几个 Windows vista 机器上截取下来的。这些工具与 Windows XP 或更新的版本（包括 Windows server 2008 和 Windows 7）差别不大。然而，在 Windows 7 中，使用这些工具的方式有所不同，而且每个工具间都有所不同。

事实上，Windows 管理员有很多工具可以利用。在这里不会介绍所有这些工具，将重点介绍实时监控 Windows 系统的工具。首先看一些基础报表工具。 <277>

以下是可以用于诊断和监控 Windows 性能问题的最流行的工具：

- Windows Experience Index
- System health Report
- Event Viewer
- Task Manager
- Reliability Monitor
- Performance Monitor

有关微软 Windows 性能、工具、技术和文档的优秀信息资源可以在微软 Technet 网站上找到。

Windows Experience

相对于微软硬件性能指标的期望而言，如果想快速了解系统性能如何，可以运行 Windows Experience 报告。

要启动 Windows Experience 报告，先单击 Start，选择控制面板上的 System，然后选择 Maintenance 下的 Performance Information and Tool。需要通过用户账户控制才能继续后面的操作。

也可以通过 Start 菜单中的搜索功能来访问 System Health Report。单击“Start”，并在搜索框中填入“performance”，然后单击链接“performance information and tools”。单击高级工具，然后单击对话框底部的“generate a system health report”链接。需要通过用户账户控制才能继续后面的操作。



微软已经改变了 Windows 7 中的 Windows Experience。该报告与之前的 Windows 版本非常相似，但是它提供了更多可以用来判断系统性能的信息。

这个报告在安装后只运行一次，但是可以通过单击 Update My Score 重新生成报告。

这个报告评估了系统性能的 5 个方面：处理器（CPU）、内存、视频控制器（图形）、视频图形加速器（游戏图形）和主要硬盘驱动器。图 7-24 显示了 Windows Experience 报告的范例。

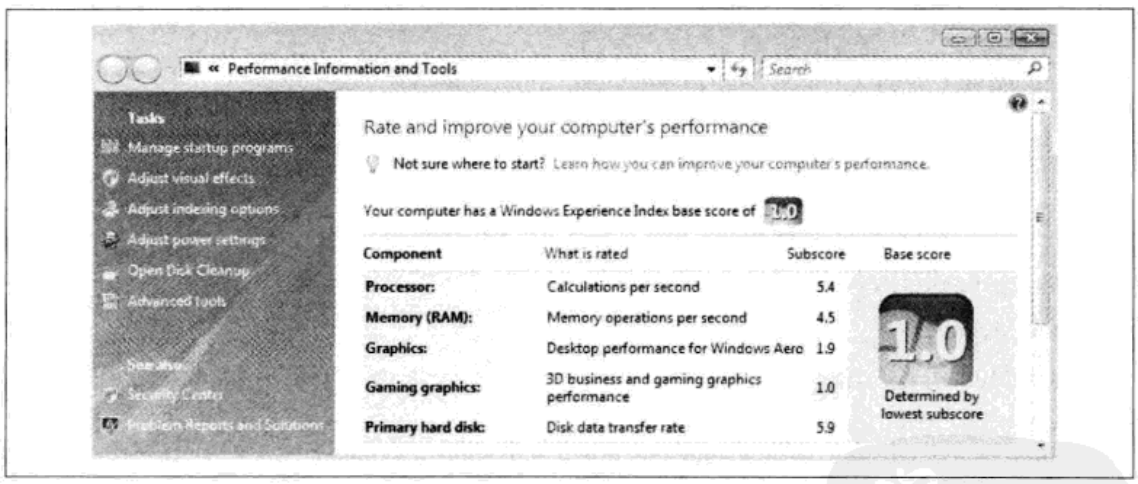


图7-24：Windows Experience 报告

278 这个报告有一个鲜为人知的功能，你可能会发现它的价值。单击“Learn how you can improve your computer's performance”链接，可以获得提高这些 Score 的最佳实践列表。



在每次重新修改系统配置时，应该运行该报告，重新生成指标。这将帮助你认识哪些配置的改变影响了系统性能。

该工具的最大用处是：在无须分析大量的指标的情况下获取系统如何运行的整体印象。任何类别中的一个低分值都可以代表一个性能问题。例如，如果你查看图 7-24 中的报告，将发现该系统有非常低的 graphics 和 gaming graphics 分值。这对于作为虚拟机或 headless server 来运行的 Windows 系统而言，是不可避免的，但是这可能会让刚刚为高端游戏系统掏出几千美元的人感到不快。

System Health Report

Windows Vista 和 Windows 7 的一个独特功能和诊断改进是：能够产生一个系统中所有硬件、软件和性能指标的快照的报告。这与 Mac OS X 的 System Profile 有些类似，而且还包含性能计数器。

要启动 System Health Report，首先单击 Start，然后选择 Control panel → System and Maintenance → Performance Information and Tools. 下一步，选择 Advanced Tools。然后单击对话框底部的链接“Generate a system health report”。需要通过 UAC 才能继续后面的操作。 279

也可以使用开始菜单上的搜索功能访问 System Health Report。单击 Start，然后在搜索框中输入“performance”，紧接着单击 Performance Information and Tools。单击 Advanced Tools，然后选择对话框底部的链接“Generate a System Health Report”。另一种访问 System Health Report 的方法是使用 Start 菜单中的搜索功能。单击 Start，然后在搜索框中输入“system health report”，紧接着单击 Start 菜单中的相关链接。需要通过 UAC 才能继续后面的操作。图 7-25 显示了 System Health Report 的一个实例。

该报告包含了有关系统的所有信息——所有的硬件、软件和有关系统的其他方面都记录在该报告上。请注意这个报告被分为几部分，可以展开或折叠报告以方便查看。下面的列表简单介绍了每个部分所显示的信息。

- *System Diagnostic Report*
 系统的名称和报告生成的日期。
- *Diagnostic Results*
 报告运行中所产生的警告信息，确定计算机存在的潜在问题区域。此外，还包括报告运行阶段的系统性能的简要概述。
- *Software Configutation*
 系统上安装的所有软件的清单，包含系统安全设置、系统服务和启动程序。
- *Hardware Configuration*
 有关磁盘、CPU 性能计数器、BIOS 信息和设备的重要元数据列表。

- *CPU*
 报告时期运行的进程列表，以及有关系统组件和服务的元数据列表。
- *NetWork*
 系统上的网络接口和协议的元数据。
- *Disk*
 所有磁盘设备的性能计数器和元数据。
- *Memory*
 内存的性能计数器，包括进程列表和内存使用率。
- *Report Statistics*
 报告运行时期的系统的一般信息，如处理器速度和安装的内存容量。

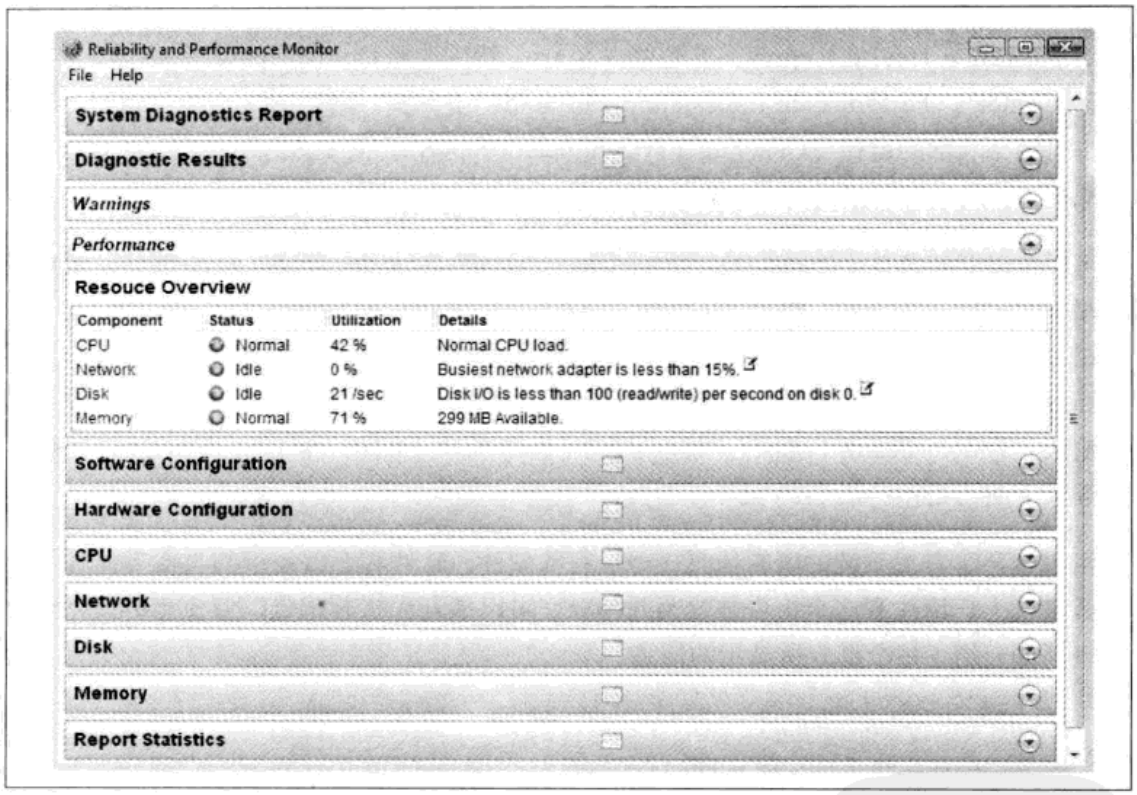


图7-25：System Health Report

280 System Health Report 是快速了解系统如何配置和运行的关键。它是一个静态报告，并显示了系统的快照。

在 Hardware Configuration、CPU、Network、Disk 和 Memory 部分显示了大量细节信息。欢迎探索这些部分以获取系统的更多细节信息。

除了检测性能计数器以外，这个工具的最大用处是：存储报告，并在日后系统运行很差

时将它与其他报告相比较。可以通过选择 File → Save As 将该报告存储为 HTML 版本。

可以将被保存的报告作为系统性能的基准。如果多次在低、中和高使用率情况下生成系统报告，应该可以将它们放在一起形成系统性能的一般期望。这些期望很重要，因为可以通过它们确认系统性能问题是否在期望范围内。当一个系统在一段时间内进入一段不寻常的高负荷期，而此时又希望系统处在低负荷时，用户可能会投诉系统问题。如果你有这些报告可以进行比对，则可以节约大量调查系统变慢的确切根源的时间。

Event Viewer（事件查看器）

281

Windows Event Viewer记录了应用程序、安全性和系统事件的所有信息。这是已发生的(或继续发生的）事件的巨大信息来源，它应该是用于诊断和监控系统的主要工具之一。

可以使用 Event Viewer 完成大量事情。例如，可以生成任何日志的自定义显示形式，并为今后的诊断保存日志，且为将来的特殊事件设立警报。我们将集中查看日志。如需了解有关 Event Viewer 的更多信息，以及怎样设置自定义报告和订阅事件，请参看 Windows 帮助文档。

要启动 Event Viewer，单击 Start 按钮，紧接着单击右键并选择 Manage。必须通过 UAC 才能继续后面的操作。然后你可以单击左侧面板中的 Event Viewer 链接。还可以通过以下方法启动 Event Viewer：单击 Start，输入“event viewer”，并按“Enter”键。

该对话框有三个默认的窗口。左边的窗口是一个含有 custom、logfile、applications 和 services logs 的树形视图。日志被显示在中间窗口，而右边窗口包含 Action 菜单项。日志条目默认按时间和日期降序排列。这使你可以首先看到最近的消息。



只要你喜欢，可以自定义 Event Viewer 视图。甚至可以通过单击日志的列头对日志进行分组和排序。

打开 Windows 日志的树形菜单，并查看应用程序、安全和系统（包括其他部分）的基本日志文件。图 7-26 显示了 Event Viewer 中展开和扩大的日志树。

这些日志可以查看和搜索，其中包括：

- *Application*
 用户应用程序和操作系统服务生成的所有信息。当诊断应用程序的问题时，这是一个很好的查看部分。
- *Security*
 记录访问和行使权限的信息，并记录访问任何安全对象的失败尝试。这是查找与用

户名和密码问题相关的应用程序错误的好地方。

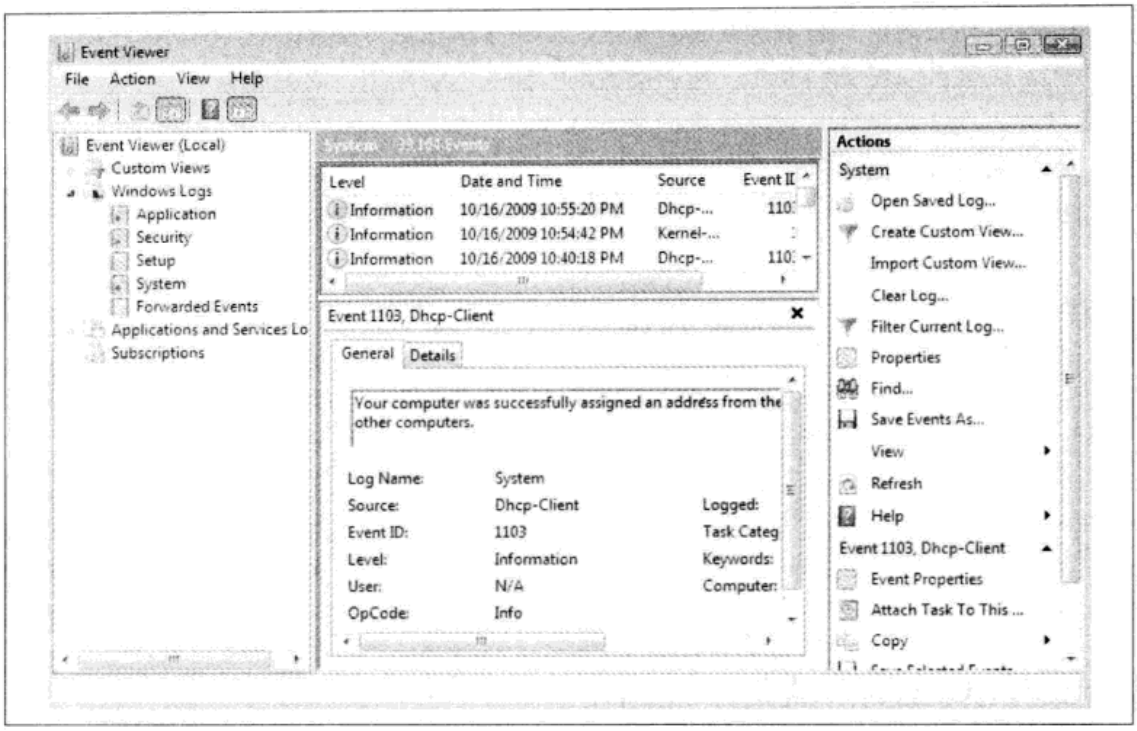


图7-26：Windows event viewer

- *Setup*

 有关程序安装的信息。这是查找错误安装或删除软件的信息的好地方。

282

- *System*

 有关设备驱动和 Windows 组件的信息。这是用于诊断整个系统或设备问题的最有用的一组日志。它包含在系统级别运行的设备的所有行为信息。

- *Forwarded Events*

 从其他计算机转发的信息。参看 Windows 文档中有关使用远程事件日志记录的详情。

挖掘这些日志可能很困难，因为它们展示的大部分信息是开发者所感兴趣的，而对于一般人可读性并不强。为方便起见，可以通过以下方法搜索任何日志：单击 Action 窗口中的 Find 操作，并输入一串字符。比如，如果你对内存问题感兴趣，可以输入“memory”，其相关内容将显示在中间窗口。

每个日志信息都可以归入以下三个类别中，这些类别适用于用户进程、系统组件和应用程序。

- *Error*

 表示一些重大错误，如错误进程、内存溢出问题或系统故障。

- *Warning*
 表示不太严重的情况或事件，如低内存或低磁盘空间。
- *Information*
 传达关于事件的数据。这通常不是什么问题，但它在诊断问题时可以提供额外信息，如删除一个 USB 驱动。

要查看日志，请打开左窗口中相对应的树。要查看任何消息的细节，请单击该消息，该消息将显示在日志条目的下面，如图 7-26 所示。在中间窗口的下部，可以单击 General 选项卡以查看消息的一般信息，如所记录的语句、什么时候发生、包含哪些日志信息和运行该进程或应用的用户。还可以单击 Detail 选项卡以查看记录数据的报告。可以以文本（Friendly View）或 XML（XML View）形式查看这些信息。还可以存储这些信息以方便以后查看，XML View 可以方便地将报告传给那些能识别该格式的工具。

Reliability Monitor

Windows 中最有趣的监控工具是 Reliability Monitor。这是一个专门用于将一段时间内发生的错误事件和重大性能事件绘制在一张图上的工具。纵轴表示每天的时间段，横轴表示当天的性能指标的总和。如果有错误或其他重大事件发生过，你将在图上看到一个红 X。坐标下方是一个下拉列表，包含软件的安装和清除信息、任何应用程序故障、硬件错误、Windows 错误和其他任何错误。

此工具对于检测一段时间内系统的性能问题是非常有用的。当过去运行正常的应用程序或系统服务开始运行缓慢时，或当系统开始产生错误信息时，它可以帮助诊断该情况产生的原因。这个工具还可以帮助找到事件第一次运行的日期，并告诉你当它运行正常时系统是如何表现的。

该工具的另外一个用途是：随着时间的推移，提供系统的一系列日常基线。这将帮助你诊断改变设备驱动的问题（Windows 管理的禁忌之一），这种变动可能直到系统明显变慢才会被发现。

总之，Reliability Monitor 让你有机会回头看看你的系统是如何运行的。而且最好的是它不必要手工启动，可以自动执行，并从日志中提取大量数据。从此，就可以自动了解你的系统的历史。



Windows 中的一个大问题是硬件的连接和配置。这里不会讨论这个问题，否则可能要写成一本书才能谈完。如果你有硬件和驱动程序的问题，可以参考 Ed Bottet al 编写的“*Microsoft Windows XP Inside Out*”（微软出版社）。

可以通过以下方式访问 Reliability Monitor：单击 Start，输入“reliability”，然后按回车键，或者单击 Reliability and Performance Monitor 链接，紧接着通过 UAC 认证后，单击右侧的树形面板中的 Reliability Monitor 链接即可。图 7-27 显示了 Reliability Monitor 的一个实例。



在 Windows7 中，可以通过以下方式启动 Reliability Monitor：单击 Start 按钮，紧接着在搜索框中输入“action center”，然后按下回车键即可。启动 Reliability Monitor 后，可以选择 Maintenance → View reliability report。这个版本的报告虽然与 Windows 以前的版本有所不同，但它以简洁的包的方式提供了相同的信息。例如，新的 Reliability Monitor 报告将已知事件列表以单一的列表形式显示，而不是以下拉菜单的形式显示。

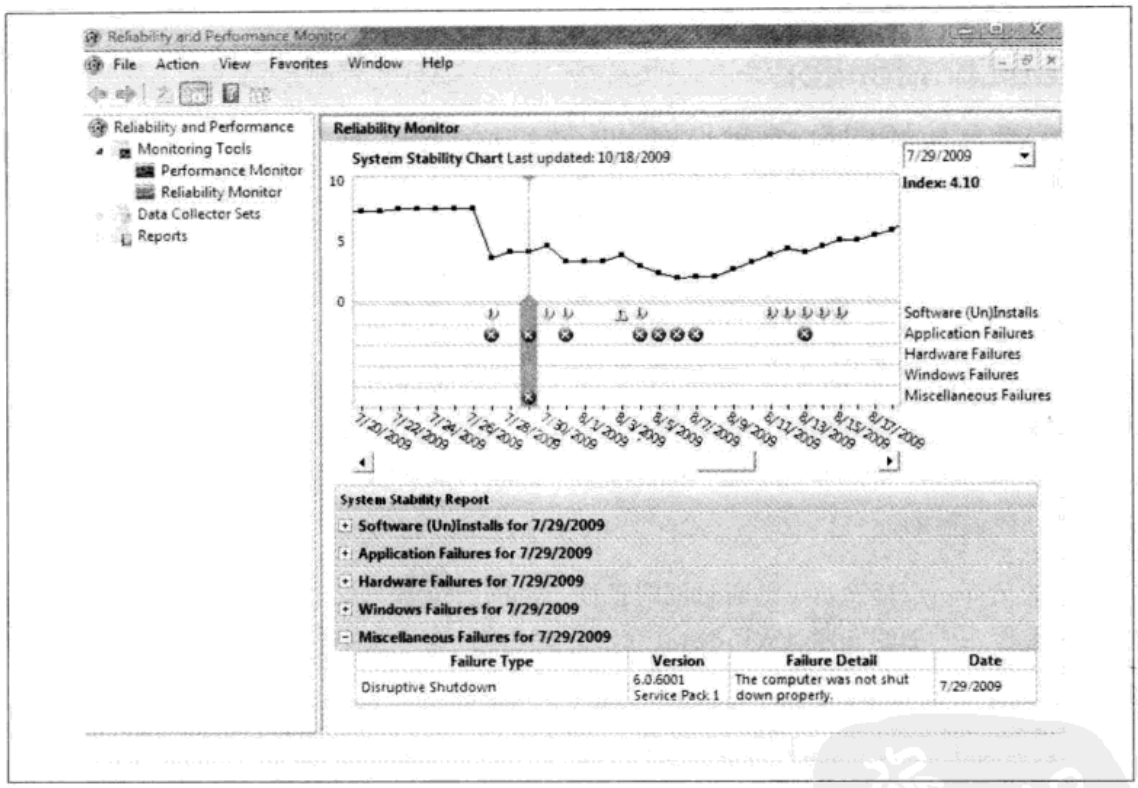


图7-27：Reliability Monitor

285

The Task Manager（任务管理器）

Windows 任务管理器用来显示正在运行的进程的动态列表，它已经存在了好长时间，且在 Windows 的各个版本中得到改善。

任务管理器提供了一个标签对话框用于显示运行的应用程序、进程（这类似于 Linux 的

top 命令)、系统上的服务活动、CPU 性能、网络性能和用户列表。与其他的报告不同，该工具动态地产生数据并定期刷新。因此，在低性能时期，这个工具是用于观察系统的非常好的工具。

这个报告虽然与 System Health Report 显示相同的信息，但是该报告的信息显示的信息更紧凑且信息在不断更新。可以在该报告中找到用于诊断性能的 CPU、资源占用进程、内存和网络的所有关键指标。很明显，这个报告缺少磁盘性能报告。

任务管理器的一个有趣的功能是：它在开始菜单栏的通知区域里显示微型的性能指标。这让你有机会监视系统的使用峰值。



运行动态性能监控工具将会消耗系统资源且影响低性能系统的运行。

使用快捷键 Ctrl + Alt + Del，然后选择菜单栏中的任务管理器，即可启动任务管理器。图 7-28 是任务管理器显示进程列表的一个实例。

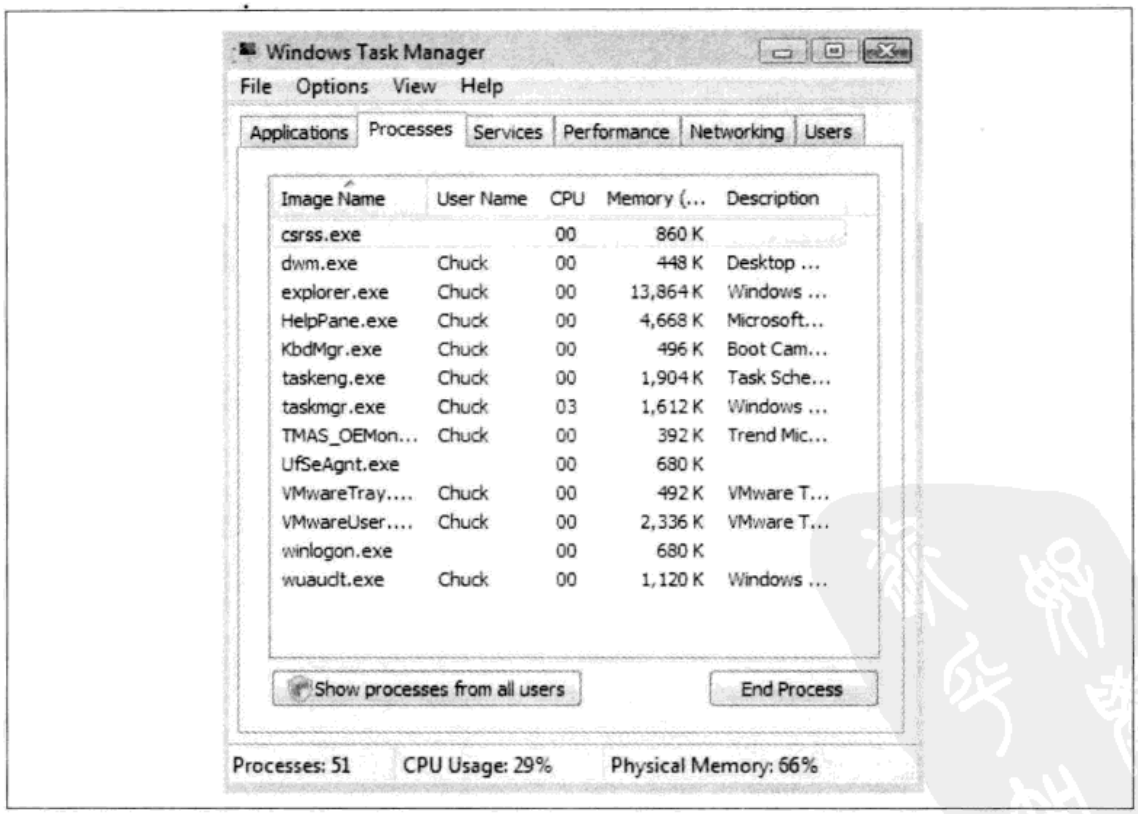


图7-28：任务管理器

Performance Monitor

Performance Monitor（性能监控器）用于跟踪 Windows 操作系统性能，它允许选择关键指标，并以图形方式显示各指标一段时间内的值，还能够保存当前信息以便日后查看，并且能够为你的系统建立基线。

性能监控器差不多包含了系统所有的指标信息，它有许多关于性能的基本领域（CPU、内存、磁盘和网络）的细节信息的计数器，还包含很多其他类别的信息。

要启动性能监控器：首先单击 Start，紧接着选择 Control Panel → System and Maintenance → Performance Information and Tools。然后单击高级工具，再单击对话框中间附近的链接 Open Reliability and Performance Monitor。通过 UAC 认证后，单击左侧树形菜单中的 Reliability Monitor 即可访问性能监控器的相关功能。

286 也可以通过以下方法启动性能监控器：单击 Start 按钮，输入“reliability”并按回车键或者单击 Reliability and Performance Monitor 链接。通过 UAC 认证后，单击左侧树形菜单中的 Reliability Monitor 即可访问性能监控器的相关功能。图 7-29 显示了性能监控器的图例。

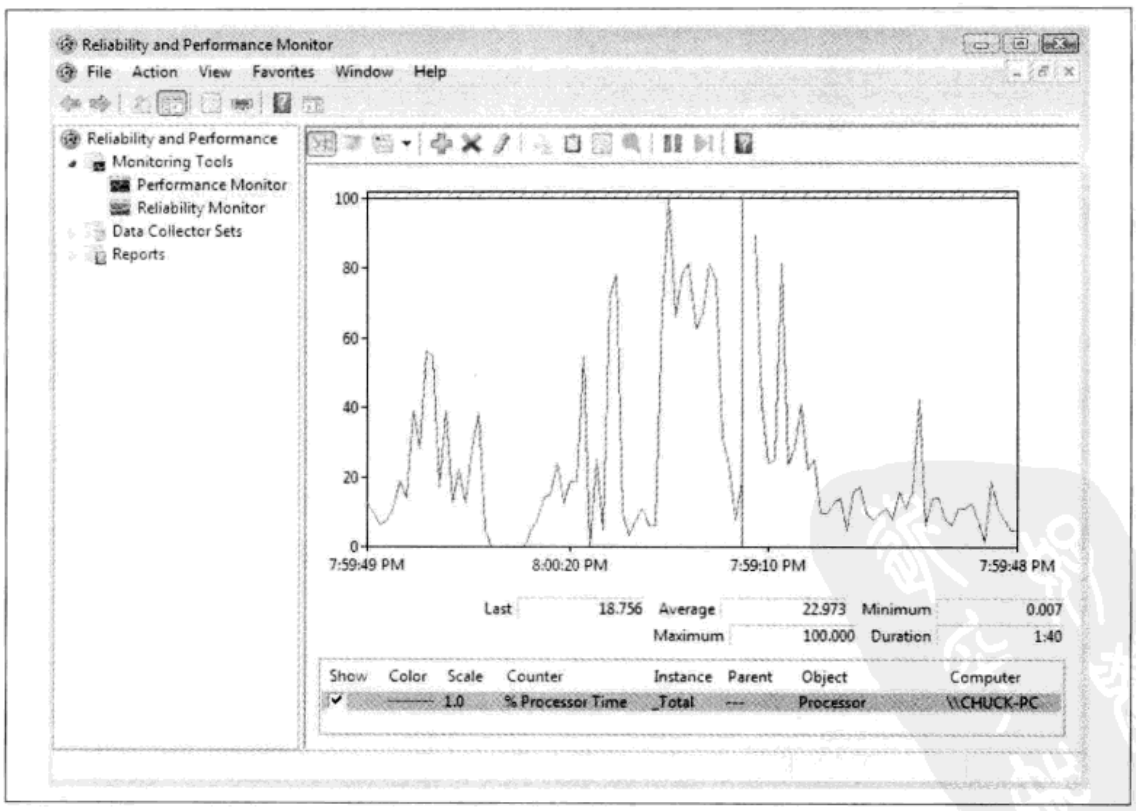


图7-29：性能监控

微软有两个级别的指标：其中一个是对象，它提供了各领域（如处理器或内存）的高级视图，另一个是计数器，它显示了系统的具体细节信息。因此，既可以监控 CPU 的整体性能，也可以查看其细节信息，如闲置时间百分比或用户进程数量。可以通过单击工具栏上的绿色加号将这些对象或计数器添加到主显示图上。该操作将会弹出一个对话框，让你从一个长清单中选择需要添加到图中的项。添加项是个简单的过程：先选择对象，然后展开右侧的下拉列表，并将所需的对象拖曳到右侧的列表中。

可以添加很多需要查看的项，显示图表的轴将根据添加的项相应地做出变动。如果你添加的项过多，或者各项间的值差别太大，这个显示图表将变得不可靠。最好的做法是：每次选择几个相关的项（如只有内存计数器）进行查看，这样所显示的图表比较有意义且方便查看。 287

关于性能监控器的详细、全面的介绍超出了本章的范围。建议去查阅它的附加功能，如数据收集器集合和改变图表显示规格参数表。有许多描述这些功能详细情况的文档。

性能监控器的功能多种多样，使其成为形成系统基准和记录系统一段时间内的状况的最佳选择，可以将它当成实时诊断工具来使用。



如果已经使用了可靠性或性能监控工具，你可能注意到一个很少被提及的特征——Resource Overview。它是可靠性和性能监控器的默认视图。它提供 CPU、磁盘、网络和内存动态性能图的显示，图表下的下拉详细信息框包含了以上四个方面的相关信息。该报告是任务管理器性能图表的扩展形式，并为 Windows 操作系统的性能监控和诊断提供了另外一个参考来源。 288

对微软 Windows 的性能监控器的简单介绍，应该能够让你相信微软 Windows 平台很难监控且缺乏先进工具的说法只是一个谎言。这些监控工具显示的信息广泛（有些甚至显示的信息过多）且提供系统数据的各种显示视图。

预防性维护监控

到目前为止所讨论的技术都是系统状态的快照。然而，大多数人都认为监控通常是自动化的任务，并且能够自动记录异常的统计数据。当异常被发现时，将以警报的形式被发送给管理员（或管理员组），从而让管理员知道系统可能出现的问题。这将使被动监控系统状态转变为主动监控。

大量第三方工具将监控、报告和预警结合成易用的接口，甚至有一个针对于整个设施的监控和预警系统。比如，Nagios 能够监控整个 IT 基础设施并为异常设立预警。

当然还存在一些作为操做系统或数据库系统的一部分或附属品的监控和预警系统。我们将在第 13 章介绍 MySQL 企业版监控。

小结

有许多介绍性能优化和安全监控的参考，本章介绍了系统监控的相关内容，简单介绍了监控操作系统和服务器性能的工具、技术和概念。下一章将开始介绍 MySQL 系统的监控任务，并讨论一些常用的做法，它们能帮助你的 MySQL 系统保持运行在最佳状态。

289

"Joel!"

Joel 熟悉这个声音和语调，他的老板正朝他的办公室走来，打算催促另一个紧急任务。他转过头去看到老板正走进他的办公室。"你有阅读 Sally 的关于运行慢的邮件吗？"

Joel 想起来 Sally 是他的一个同事，她曾经写了封邮件询问为什么她的应用程序运行得很慢。他刚刚才完成一个简单的检查，问题不在内存和磁盘上，那里有大量的内存和磁盘空间。

"是的，我正在深入调查这个问题。"

"把它当做你的首要任务，市场部门需要在最后期限内出示季度销售预测。让我知道你发现了什么。"老板点了下头然后走开了。

Joel 叹了口气，继续检查 CPU 使用报告，同时思考如何用非技术语言来描述一个技术问题。

监控MySQL

Joel 感觉今天将是美好的一天。因为一切都进行得很顺利：服务器的性能测定结果看起来不错，用户投诉也在下降。他成功地将服务器进行了重新配置并使其性能大为提高。只有一个应用程序性能仍然低下，但他确信这并不是硬件或操作系统引起的问题，它更像是由一个写得很差的查询语句所带来的问题。于是，他给老板写了封电子邮件，来解释他的发现，以及他正在忙于处理的遗留问题。

Joel 听到他的办公室外传来一阵急促的脚步声。他本能地向门外看去，等待着老板那熟悉的身影出现。Summerson 只是路过，一句话都没说仅向他点了个头，把他吓了一跳。

他耸耸肩，开始读他的电子邮件。正在这时，一个标题大写的“高优先级”的新消息出现了。这是他的老板发过来的。Joel 觉得自己过于紧张了，松了一口气后打开消息。当他读这封邮件时仿佛能在脑海里听到老板的声音。

“Joel，这些报告写得很好。我特别喜欢你写的有关内存和磁盘性能的细节。我想要你写一个关于数据库服务器的类似报告，还想要你深入了解下某个开发人员在查询语句方面带来的问题。Susan 会将相关细节发给你。”

Joel 深深叹了口气，再次打开他最喜欢的 MySQL 书，以了解更多有关监控数据库系统的知识。“我希望这本书能深入地介绍每个独立的部件。”他咕哝着，并知道自己需要快速学习 MySQL 的高级功能。

现在，你已经知道监控是如何工作的，以及如何让操作系统以最高效率运行了，那么如何知道 MySQL 服务器是否正处于最高效运行呢？或者，怎么知道它们不是高效运行的？

本章先介绍 MySQL 监控，然后介绍如何监控和提高数据库性能。最后将介绍提高数据库系统性能的最佳实践方法。

292

什么是性能

在开始讨论数据库性能及监控和优化 MySQL 服务器的常用最佳实践之前，最好先定义一下性能的含义。就本章而言，良好的性能被定义为能够满足用户的需求，也就是指系统按照用户期望的那样得当地运行，而低性能被定义成系统运行不佳。通常情况下，良好的性能意味着响应时间和吞吐量能够达到用户的期望效果。这可能看起来不是很科学，专业的管理员知道，衡量好系统的最佳指标就是让用户满意。

这并不意味着我们不测量性能。相反，我们会而且必须测量性能，以获知有什么问题需要修正，什么时候进行修正，如何修正。此外，如果定期测量性能，你甚至会预测到用户在什么情况下感到不高兴。用户不在乎你是否将缓存命中率增加三个百分点。你可能会为这样的事情感到自豪，但是，与用户体验相比，指标和数字是无意义的。

当处理性能问题时，必须采用一个非常重要且基本的原理：永远不应该调整服务器、数据库或存储引擎的参数，除非有深思熟虑的计划，并相当了解更改后的期望和后果。更重要的是，在没有评估更改所带来的影响的前提下，永远不要调整服务器参数。完全有可能发生这样的情况：你可能在短期内提高了服务器性能，但是长期会对性能产生负面影响。最后，你应该查阅各个方面的参考信息，其中包括参考手册。

现在，我们已经发出了严厉的警告，接下来关注 MySQL 服务器和数据库的监控和提高性能这两方面问题。

MySQL服务器监控

管理 MySQL 服务器属于应用程序监控范畴。这是因为绝大多数性能参数是由 MySQL 软件产生的，而不属于主操作系统的一部分。如前面所提到的，应该总是先监控基础操作系统，然后监控 MySQL，因为 MySQL 对主机操作系统的性能很敏感。

在 MySQL 的在线参考手册中有一整章“优化”介绍监控和性能提高的各个方面。参看 *http://dev.mysql.com/doc/refman/5.5/en/optimization.html* 以获取更多的详情。我们将讨论监控 MySQL 服务器的一般方法，并评价各种可用的监控工具，而不再重复介绍参考文档上的知识。

293

本节将介绍监控 MySQL 服务器的详情。首先简单介绍如何更改和监控系统的行为，然后讨论用于诊断性能问题和形成性能基准的监控。还将介绍诊断性能问题的最佳实践，

并介绍一些监控 MySQL 存储引擎方面的知识，这方面的知识在其他参考资料上很少被提及。

如何显示MySQL性能

可以使用两种机制来管理和监控 MySQL 服务器的运行情况。使用服务器变量来控制其运行情况，并使用服务器状态变量读取其运行情况配置和关于功能和性能的统计信息。

服务器中有很多配置变量。有些变量只能在启动时设置（被称为启动选项，这些也可以在选项文件中设置）。其他的一些变量可以被设置成适用于全局级（对于所有连接有效）、会话级别（单个连接）或同时适用于全局和会话两个级别。



会话变量只在当前连接中有效，在连接关闭时被重置。

可以使用以下命令读取服务器变量：

```
SHOW [GLOBAL | SESSION] VARIABLES;
```

可以使用以下命令更改非静态（只读）变量（可以用“,”分隔符在一条命令上同时进行多个变量的设置）：

```
SET [GLOBAL | SESSION] <variable_name> = <value>;
SET [@@global. | @@session. | @@]<variable_name> = <value>;
```

可以使用以下命令读取状态变量。前两条命令显示所有本地或会话范围（默认情况下是会话）的变量值。第三条命令显示全局范围的变量：

```
SHOW STATUS;
SHOW SESSION STATUS;
SHOW GLOBAL STATUS;
```

后面的章节将讨论如何及何时使用这些命令。

性能监控

MySQL 的性能监控是前面命令的应用。具体而言，就是设置和读取系统变量及读取状态变量。SHOW 和 SET 命令是仅有的两个可以用于监控 MySQL 服务器的工具。

事实上，有很多工具可以用于监控 MySQL 服务器。在标准发行版中可用的监控工具仅限于控制台工具，其中包括可以在 MySQL 客户端执行的特殊命令（如 SHOW STATUS）和可以从命令行运行的实用工具（如 mysqladmin）。

294



MySQL 客户端工具有时被称为 MySQL 监控器，但是不要将它与监控工具混为一谈。

还有一些 GUI 工具使监控变得更容易，如果你有这方面的需求，可以考虑选择这种工具。还可以下载 MySQL GUI 工具，其中包括高级工具，这些高级工具可以用于监控系统、管理查询和从其他数据库系统迁移数据。

下面首先介绍如何使用这些 SQL 命令，然后讨论 MySQL 管理器图形工具和查询浏览器。还将简单介绍最容易被忽视的管理工具之一——服务器日志。

一些专业的管理员在管理服务器时，可能第一时间考虑最重要的工具——服务器日志。虽然它们不如性能监控工具那么重要，但是它们在诊断性能问题时也是非常重要的。

SQL命令

所有的 SQL 监控命令都是 SHOW 命令的变体，它们显示系统及其子系统的内部信息。虽然 SHOW 命令的形式很多，以下命令列表是在监控 MySQL 服务器时使用最多的 SQL 命令：

- `SHOW INDEX FROM <table>`
 显示指定表的索引基数（cardinality）统计信息，在优化程序中使用它评估连接选择性（即索引列中非重复值）。这个命令有助于诊断性能低下的查询，尤其是查询是否使用了可用的索引。
- `SHOW PLUGINS`
 显示所有已知插件的列表。它显示插件的名称和当前状态。MySQL 最新发行版中的存储引擎是以插件形式实现的。使用这个命令获取当前可用插件及其状态的快照。
- `SHOW [FULL] PROCESSLIST`
 显示系统上运行的所有线程（包括连接）。这个命令与主操作系统的进程命令类似。显示的信息包括命令执行时的连接数据、执行时间和当前状态。像操作系统命令一样，它可以诊断不良的响应（太多线程）、僵尸进程（zombie process）（未响应或长时间运行），或者甚至诊断连接问题。当在处理低性能或未响应的线程时，使用 KILL 命令终止它们。默认行为是显示当前用户的进程。命令中使用 FULL 关键字则显示所有进程。

295



必须有全局 SUPER 权限才能查看运行在系统上的所有进程。

- SHOW [*GLOBAL* | *SESSION*] STATUS

 显示所有系统变量的值。相对其他命令而言，你可能更频繁地使用这个命令。使用这个命令读取在服务器上可获取的所有统计信息。与GLOBAL或SESSION关键字结合使用，可以选择性地只查看全局变量的统计信息或会话变量的统计信息。

- SHOW TABLE [FROM <*db*>] STATUS

 显示给定数据库的表的详情，其中包括存储引擎、排序规则（Collation）、创建数据、索引数据和行统计信息。可以将这个命令与SHOW INDEX命令结合使用，在诊断低性能查询时检查表信息。

- SHOW [*GLOBAL* | *SESSION*] VARIABLES

 显示系统变量。一般是服务器的配置选项，虽然它不显示统计信息，但是在确定当前配置是否已被更改或某些选项是否被设置时，查看变量是非常重要的。有些变量是只读的，且只可以通过配置文件或命令行在启动的时候更改，而其他的一些变量可以在全局范围内更改或在本地设置。可以将这个命令与GLOBAL或SESSION关键字结合使用，从而选择性地查看全局变量或会话变量。

限定SHOW命令的输出结果

MySQL中的SHOW命令功能很强大。然而，它经常显示过多的信息。尤其是SHOW STATUS和SHOW VARIABLES命令。

为了查看较少的信息，可以使用LIKE<*pattern*>从句，这将允许你只查看那些匹配具体正则表达式的行。最常用的例子是使用LIKE语句仅查看某些子集的变量，如复制或日志。可以在LIKE语句中使用标准MySQL正则表达式符号和控制符，这与SELECT查询中的LIKE使用方法相同。

例如，下面显示了包含“log”的状态变量：

296

```
mysql> SHOW SESSION STATUS LIKE '%log%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| Binlog_cache_disk_use    | 0     |
| Binlog_cache_use         | 0     |
| Com_binlog               | 0     |
| Com_purge_bup_log        | 0     |
| Com_show_binlog_events   | 0     |
| Com_show_binlogs         | 0     |
| Com_show_engine_logs     | 0     |
| Com_show_relaylog_events | 0     |
| Tc_log_max_pages_used    | 0     |
```

```
| Tc_log_page_size          | 0     |
| Tc_log_page_waits         | 0     |
+---------------------------+-------+
11 rows in set (0.11 sec)
```

与存储引擎相关的命令如下所示：

- SHOW ENGINE <*engine_name*> LOGS
 显示指定存储引擎的日志信息，这些信息是由存储引擎提供的，且在优化存储引擎时是非常有用的，有些存储引擎不提供这些信息。
- SHOW ENGINE <*engine_name*> STATUS
 显示指定存储引擎的状态信息，这些信息是存储引擎提供的。有些存储引擎比其他的存储引擎提供更多或更少的信息。例如，InnoDB 存储引擎显示很多状态变量，而 NDB 存储引擎只显示少许的状态变量。MyISAM 存储引擎则不显示任何信息。这个命令是查看一个给定存储引擎的统计信息的基本方法，且它在优化某些存储引擎时非常重要（如 InnoDB）。



SHOW ENGINE 命令的旧同义形式（SHOW <*engine*> LOGS 和 SHOW <*engine*> STATUS）已经过时了。

- SHOW ENGINES
 显示 MySQL 发行版的所有可用存储引擎的列表及其状态（如存储引擎是否可用）。这个命令有助于决定在数据库上使用何种存储引擎，并有助于决定复制过程中是否在 Master 和 Slave 上存在相同的存储引擎。

297 与 MySQL 复制相关的具体命令如下所示：

- SHOW BINLOG EVENTS [IN '<*log_file*>'] [FROM <*pos*>] [LIMIT [<*offset*>,] <*row count*>]
 显示被记录到二进制日志的事件。可以指定一个二进制文件用于审查（系统默认使用当前二进制文件），并限定输出从日志中某个特定位置到日志结尾的所有事件，或者限定输出从日志第一行到第 *N* 行的事件。这个命令是用于诊断复制问题的主要命令，当事件中断复制或在复制过程中导致错误时，这个命令非常有用。



如果你不使用 LIMIT 语句，且你的服务器正在运行并已经将事件记录到日志一段时间了，将会获得一个很长的输出结果。如果需要查看大量事件，应该考虑使用 mysqlbinlog 实用工具代替这个命令。

- SHOW BINARY LOGS

 显示服务器上的二进制列表。使用这个命令获取过去和当前 binlog 文件名的相关信息。每个文件的大小也被显示出来了。它是诊断复制问题的另一个有用的命令，因为它将允许你在指定的文件上使用 SHOW BINLOG EVENTS 命令，从而减少在诊断问题时必须查看的数据量。SHOW MASTER LOGS 是它的同义词。
- SHOW RELAYLOG EVENTS [IN '*<log_file>*'] [FROM *<pos>*] [LIMIT [*<offset>*,]*<row count>*]

 在 MySQL 5.5.0 中可用，这个命令和 SHOW BINLOG EVENTS 命令的功能类似，仅限制查询 Slave 上的中继日志。如果不提供日志文件名，该命令将显示第一个中继日志中的事件。这个命令在 Master 上运行无效。
- SHOW MASTER STATUS

 显示 Master 的当前配置。它显示当前二进制日志文件、文件的当前位置和所有的排他性或包容性复制的设置。当连接或重新连接 Slave 时使用这个命令。
- SHOW SLAVE HOSTS

 使用 --report-host 选项显示连接到 Master 的 Slave 列表，根据这个列表可以确定哪些 Slave 连接到 Master 上。
- SHOW SLAVE STATUS

 显示复制中 Slave 的系统状态信息。这个命令是追踪 Slave 性能和状态的主要命令，显示了大量维护 Slave 健康状态的重要信息。第 2 章介绍了更多有关这个命令的信息。

这个列表中的两个最重要的命令是 SHOW VARIABLES 和 SHOW STATUS。这里有许多变量（大约有 290 个状态变量），因此一旦你掌握 LIKE 从句，就可以将结果精准到系统特定方面的信息。 298

变量列表通常是按字母表顺序排列的，并常常按功能分组。然而，有时变量没有整齐地排序。在这种情况下，可以使用关键字查找它们。例如，SHOW STATUS LIKE '%thread%' 显示所有与线程执行相关的状态变量。例 8-1 显示在 MySQL 最新测试版上运行这个命令的结果。

例8-1：显示线程状态变量

```
mysql> SHOW VARIABLES LIKE '%thread%';
+----------------------------+--------------------------+
| Variable_name              | Value                    |
+----------------------------+--------------------------+
| innodb_file_io_threads     | 4                        |
| innodb_read_io_threads     | 4                        |
| innodb_thread_concurrency  | 0                        |
| innodb_thread_sleep_delay  | 10000                    |
| innodb_write_io_threads    | 4                        |
```

```
| max_delayed_threads         | 20                          |
| max_insert_delayed_threads | 20                          |
| myisam_repair_threads       | 1                           |
| pseudo_thread_id            | 1                           |
| thread_cache_size           | 0                           |
| thread_handling             | one-thread-per-connection |
| thread_stack                | 262144                      |
+----------------------------+----------------------------+
12 rows in set (0.00 sec)
```

这个例子不仅显示了那些用于线程管理的状态变量，还显示了用于控制InnoDB存储引擎的线程。虽然有时候将获得比你想象的更多的信息，但是可以使用LIKE语句查找指定的变量。

知道更改哪些变量和监控哪些变量是监控MySQL服务器最具挑战性的部分。MySQL在线参考手册上包含了有关这方面的大量有价值的信息。

为了阐明MySQL服务器上可以被监控的功能，我们讨论控制Query Cache的变量。如果你在应用数据中使用MyISAM存储引擎，Query Cache是MySQL的一个最重要的性能特征之一。它允许服务器在内存中缓存频繁使用的查询语句和查询结果。因此，一个查询运行得越频繁，这个查询的结果就越可能从缓存中读取，而不是重新审查索引结构和表去检索数据。显然，从内存中读取数据比从硬盘上读取数据要快很多。如果你的数据读取的频率比写入（更新）频率高，这可以提高性能。

299 每次运行查询，这个查询将被放入缓存并且有生命周期，该生命周期由它最近被使用的情况（旧查询首先被回收）和Query Cache可用的内存量来决定，另外，有大量事件可以将查询从缓存中删除。这里列举了部分事件：

- 频繁更改（数据或索引）。
- 不同形式的查询，这将会导致缓存命中率丢失。因此，使用标准查询去正常访问数据是非常重要的。你将在本章后面看到视图在这个方面的作用。
- 当查询从临时表中获取数据时。
- 事务事件可以使内存中的查询无效（如COMMIT）。

可以通过检查have_query_cache变量来确定MySQL安装程序中的Query Cache是否被配置和是否可用。这是个系统全局变量，但是它是只读的。可以使用多个变量中的一个来控制Query Cache。例8-2显示了Query Cache的服务器变量。

例8-2：Query Cache服务器变量

```
mysql> SHOW VARIABLES LIKE '%query_cache%';
+------------------------------+----------+
| Variable_name                | Value    |
```

```
+------------------------------+----------+
| have_query_cache             | YES      |
| query_cache_limit            | 1048576  |
| query_cache_min_res_unit     | 4096     |
| query_cache_size             | 33554432 |
| query_cache_type             | ON       |
| query_cache_wlock_invalidate | OFF      |
+------------------------------+----------+
6 rows in set (0.00 sec)
```

正如你所见，可以更改很多东西去影响 Query Cache。最值得注意的是通过设置 query_cache_size 变量去临时关闭 Query Cache——这个变量用于设置 Query Cache 的可用内存大小。如果将这个变量设置为 0，它将迅速关闭 Query Cache。这与 have_query_cache 变量无关，它仅仅表明这个特性可用。为了获取更多关于配置 Query Cache 的信息，请查看 MySQL 在线参考手册中的“Query Cache 配置”这一章。

可以通过检查多个状态变量来观察 Query Cache 的性能，如例 8-3 所示。

例8-3：Query Cache状态变量

300

```
mysql> SHOW STATUS LIKE '%Qcache%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Qcache_free_blocks      | 0     |
| Qcache_free_memory      | 0     |
| Qcache_hits             | 0     |
| Qcache_inserts          | 0     |
| Qcache_lowmem_prunes    | 0     |
| Qcache_not_cached       | 0     |
| Qcache_queries_in_cache | 0     |
| Qcache_total_blocks     | 0     |
+-------------------------+-------+
8 rows in set (0.00 sec)
```

在这里我们将看到 MySQL 服务器中比较不可思议的一种不一致性。可以使用以 query_cache 开头的变量来控制 Query Cache，但是状态变量是以 Qcache 开头的。虽然这个不一致性是故意设计成的（有助于区分服务器变量和状态变量），但像这样奇怪的设计将会使搜索正确的项复杂化。



可以并应常定期使用 FLUSH QUERY CACHE 命令重整 Query Cache。这样不会删除内存中的结果，但是允许重新组织内存以更好地使用内存。

允许你管理和配置并监控其性能的 Query Cache 间有很多细微的区别。这使得 Query

Cache 成为说明监控 MySQL 服务器的复杂性的好例子。没有单独章节可以覆盖所有这样的话题。因此，本章描述的这些实例通常是为使用 MySQL 的功能而设计的。具体详情可能需要额外的研究和深入阅读 MySQL 在线参考手册。

另外两个用于监控复制的非常有用的命令是 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令。我们将在下面的章节介绍这些命令的详情。

mysqladmin实用工具

mysqladmin 命令行实用工具是命令行工具套件中的重量级工具。这个工具可以执行很多选项和工具（被称为命令）。MySQL 在线手册简单讨论了 mysqladmin。本节将讨论监控 MySQL 服务器的选项和工具。

301 因为这个实用工具是从命令行启动运行的，它使得管理员可以编写一系列操作脚本，这比直接运行 SQL 命令要容易很多。事实上，有些第三方监控工具将 mysqladmin 和 SQL 命令结合使用以用于收集信息，并以其他形式显示。

必须指定连接到运行中的服务器的连接信息（用户、密码、主机等）。以下列表显示了常用的命令。如你所见，这些命令中的大多数与 SQL 命令等效，产生相同的信息。

- Status
 简要显示服务器状态信息，包括正常运行时间、线程数（连接）、查询数和一般的统计数据。这个命令提供服务器健康状况的一个快照。
- extended-status
 显示系统统计信息的完整列表，与 SQL SHOW STATUS 命令类似。
- processlist
 显示当前进程的列表，并与 SQL SHOW PROCESSLIST 命令类似。
- kill *<thread id>*
 杀死一个指定的线程。它与 processlist 结合使用，可以帮助管理失控或悬停的进程。
- variables
 显示系统服务器变量和值。与 SQL SHOW VARIABLES 命令类似。

还有很多其他的选项和命令没有在这里罗列出来，其中包括在复制过程中启动和停止 Slave 的命令，还有用于管理各种系统日志的命令。

mysqladmin 的最佳功能之一是对一段时间内的信息的比较。--sleep*n* 选项告诉实用程序每隔 *n* 秒执行一次指定的命令。例如，使用如下命令，让本地主机上的进程列表每三秒刷新一次：

```
mysqladmin -uroot --password --socket=<sock> processlist --sleep 3
```

这个命令将一直执行，直到你使用 Ctrl+C 快捷键关闭工具为止。

也许最强大的功能是扩展状态命令的比较结果。使用 `--relative` 选项将先前的执行值与当前值相比较。例如，使用以下命令查看系统状态变量先前的值和当前值：

```
mysqladmin -uroot --password --socket=<sock> extended-status -relative
--sleep 3
```

还可以合并命令从而同时获得多份报告。例如，使用以下命令同时查看进程列表和状态信息：

```
mysqladmin --root ... processlist status
```

`mysqladmin` 工具还有很多其他用处，可以使用它关闭服务器、刷新日志、ping 服务器、在复制中启动和关闭 Slave，以及刷新权限表。为了了解更多关于 `mysqladmin` 工具的信息，请查看 MySQL 在线文档中的“mysqladmin——管理 MySQL 服务器的客户端”这个章节。图 8-1 显示了一个没有负载的系统的输出样本。

302

```
+----+-------------+-----------+----+---------+------+-----------------------------------+------------------+
| Id | User        | Host      | db | Command | Time | State                             | Info             |
+----+-------------+-----------+----+---------+------+-----------------------------------+------------------+
| 1  | system user |           |    | Daemon  | 0    | Waiting for event from ndbcluster |                  |
| 18 | root        | localhost |    | Query   | 0    |                                   | show processlist |
+----+-------------+-----------+----+---------+------+-----------------------------------+------------------+
Uptime: 6325  Threads: 1  Questions: 139  Slow queries: 0  Opens: 17  Flush tables: 2  Open tables: 1  Queries
per second avg: 0.21
```

图8-1：mysqladmin进程和状态报告样例

MySQL GUI工具

目前，MySQL GUI 工具被绑定到一个下载文件中，并且可以在 MySQL 网站上下载适用于不同操作系统的 MySQL GUI 工具包：

- MySQL Administrator 1.2
- MySQL Query Browser 1.2
- MySQL Migration Toolkit 1.1

后面的章节将详细讨论 MySQL Administrator 和 MySQL Query Browser。MySQL Migration Toolkit 用于自动从其他数据库系统上迁移数据模式。它可以使 MySQL 的管理和使用方便快捷。



MySQL Migration Toolkit 在 Mac OS X 平台上不可用。

MySQL管理器

MySQL 管理器几乎是个万事通，它提供以下功能：显示和更改系统变量、管理配置文件、检查服务器日志、监控状态变量，甚至以图形方式展示一些较重要特征的性能。它还有一套完整的管理选项集，这些选项用于管理用户和查看数据库配置信息。虽然它最初试图替代 `mysqladmin` 工具，但是大众需求确保在可预见的未来这两个工具将并存。

303 可以在任何平台上使用 MySQL 管理器，并且可以访问一个或多个连接到客户端的服务器。这将使该工具更方便地监控网络上的多个服务器。



MySQL 管理器的一个独特功能是分析宕机的服务器。虽然大部分功能不可用（如不存在实时性能指示器），但仍然可以查看配置信息并检查日志。它是诊断服务器崩溃的好工具。

为了连接一个宕机的服务器（被称为配置服务模型），打开这个工具，并按 Ctrl 键（Mac OS X 系统上的 Apple 键）。请注意 Connect 按钮变成了 Skip。按 Skip 键，然后这个工具将转换成配置服务模式。在这里，可以启动和停止服务器、更改启动变量和查看服务器日志。

MySQL 管理器的功能被分为 10 类，每类在应用程序中以选项卡形式展示。原则上讲，我们对显示状态变量值的工具感兴趣。MySQL 管理器有个与 `mysqladmin` 工具类似的动态功能，但是 MySQL 管理器的这个功能更高级些，它可以显示值的运行图。这些工具隶属于健康工具。如果单击 Connection Health 选项卡，将会看到如图 8-2 ～图 8-4 所示的图形。

图 8-2 分三个部分。第一个部分监控连接使用情况（服务器上目前有多少连接），这有助于确定服务器是否过载（拥有过多的连接）或者确定何时排除零星的连接。



图 8-2 ～图 8-4 是来自于 Mac OS X、Windows 7 和 Linux 系统的混合快照。如你所见，每个平台的显示都有些不同。Mac OS X 系统在菜单位置上有所不同，但是其他的功能设置相同。

304 图 8-2 中的 Traffic 图显示网络通信状况，并有助于确定网络的潜在问题。

最有趣的图是底部显示的随时间推移的 SQL 查询图，它有助于确定系统上的查询是否过载。

这里显示的实例描述了这样的系统：运行少量连接、网络通信量小和中等负载的查询，展示了一个脉冲图。

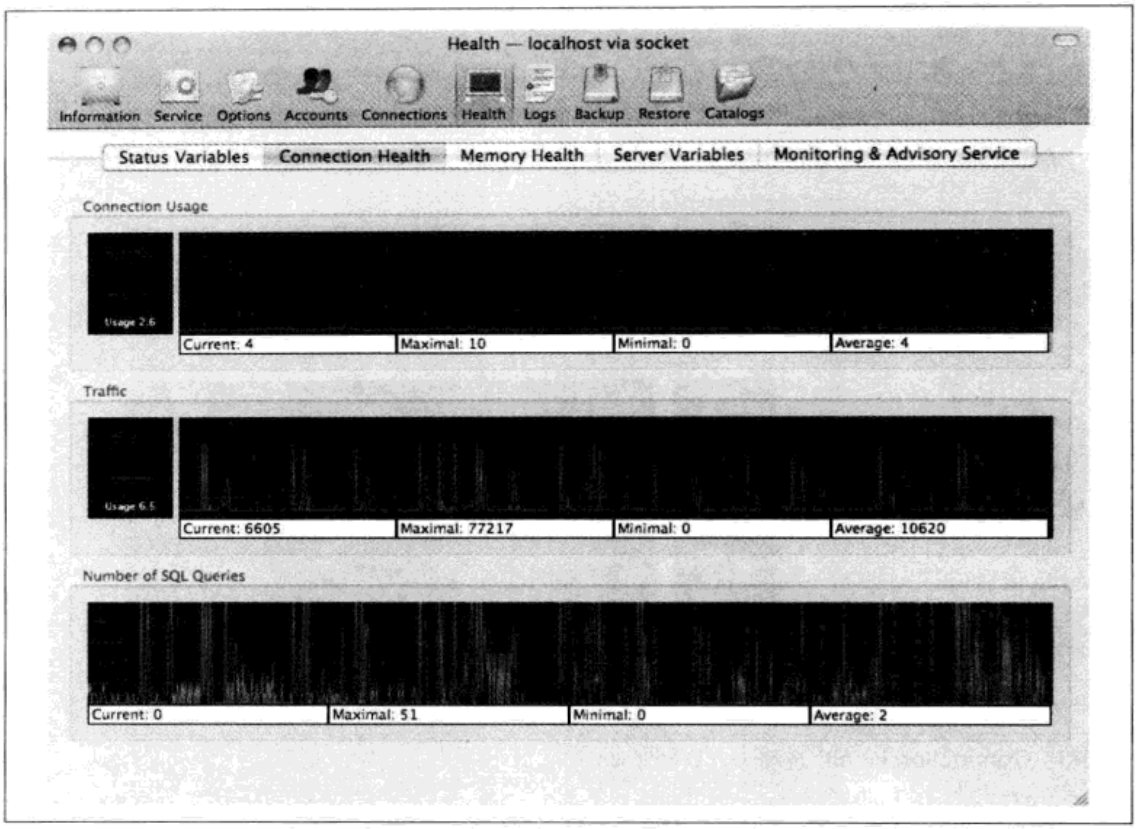


图8-2：Connection Health 选项卡（Mac OS X）

状态变量的图形显示是 MySQL 管理器的最佳功能之一。虽然这些图形掩盖了基本状态变量，仅显示一段时间内的最大值和最小值，但是这些图有助于发现潜在的问题。

长期使用这样的工具将会增加少量的开销，结果也会出现一些偏差。但是，对于一个有很多操作的系统而言，这可能不是什么大问题。

图 8-3 描述了一个基本不常用的系统，展示 Windows 平台上的 MySQL 管理器的实例。然而，这个例子作为在复制拓扑结构中需要注意的东西，是值得考虑的。特别是如果其中一个 Slave 表现这种行为，说明 Slave 出现了严重问题。同样，也有可能表明向外扩展的数据模式没有按你设计的那样运行，或者你的负载均衡策略没能成功地为服务器分配查询。

图 8-4 显示了 Linux 系统上运行的 MySQL 管理器，并演示了另外一个处于中等负载的系统的实例。请注意连接使用率和网络通信量的波形指示器显示了使用率的相对百分比，这个估量图让你一眼就能看出连接使用率和网络通信量的峰值。

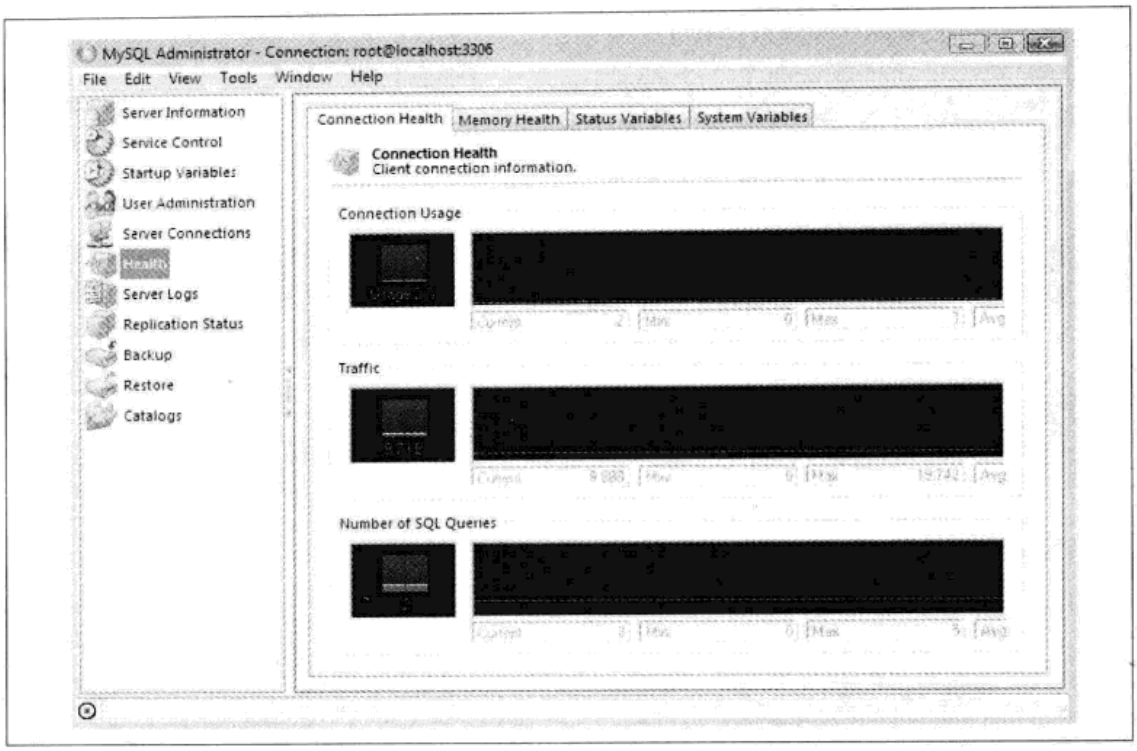


图8-3：Connection Health选项卡（Windows 7）

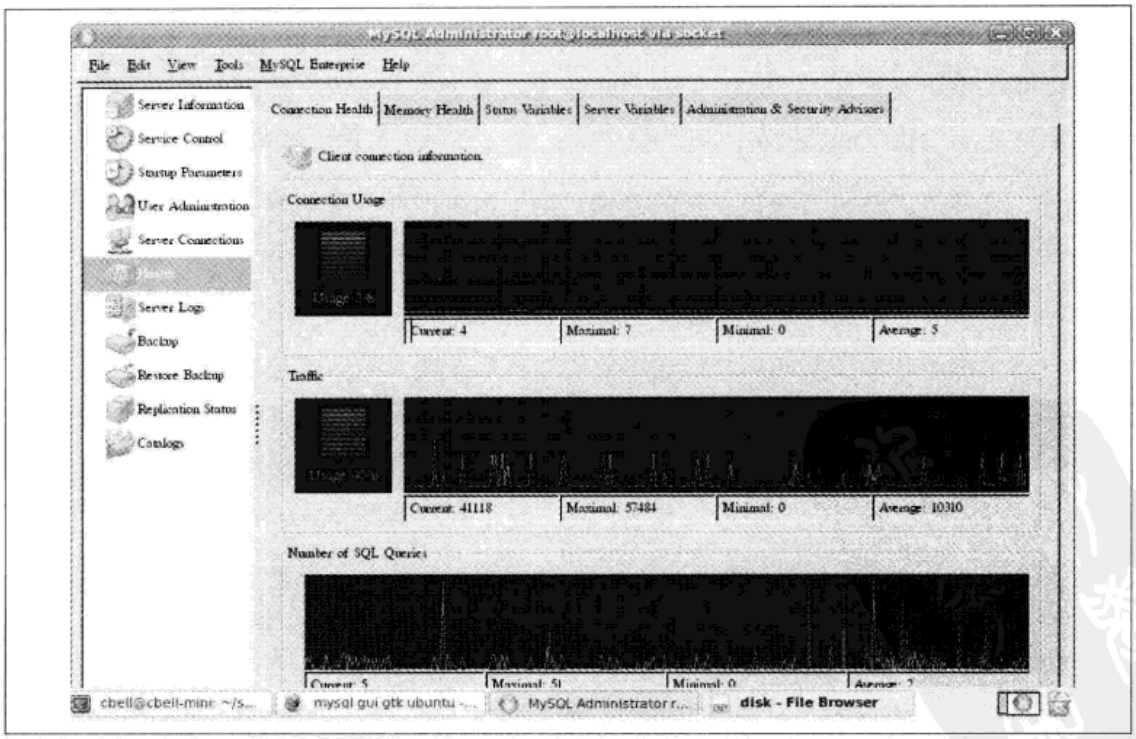


图8-4：Connection Health选项卡（Linux）

MySQL 管理器的 Memory Health 选项卡显示了 Query Cache 和 Key Cache 的健康状况动 306
态图。图 8-5 显示了中等负载的系统上运行的报告图例。在这种情况下，这个系统正在运行 MySQL 集群产品的高级诊断测试。

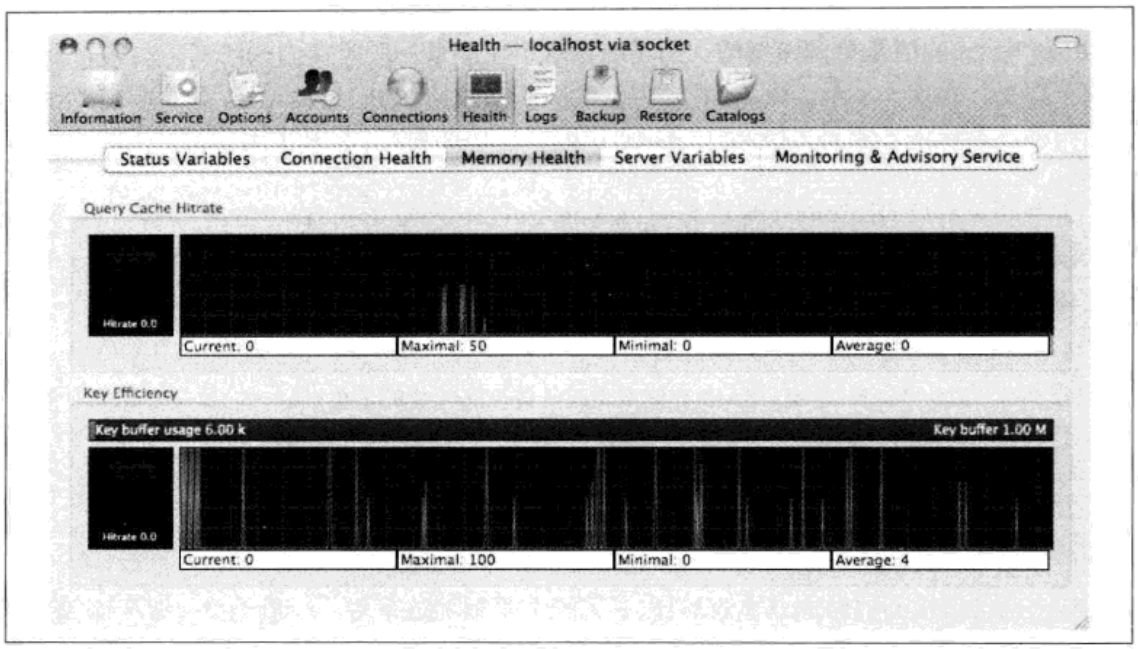


图8-5：Memory Health选项卡

与手工查询 Query Cache 状态变量有所不同，从这个显示中更容易查看 Query Cache 的性能趋势。类似于其他的动态展示，这个图也有峰值指示器，可以在值突然达到峰值时发出警告。

MySQL 中的不确定性原理

你可能比较熟悉海森堡不确定性原理，该原则表示：原子的位置越精确，那么动量不确定程度就越大，反之亦然。监控包含运行查询的 MySQL 以获取状态变量的值，因此每搜集一次数据，就会增加某些测量变量的值。因此，监控 MySQL 会有一些花费，且采样过密会减少指标值的意义。当然，这个原理一般应用在数据探测上。

你应该认识到：频繁采样会导致系统花费更多的采样时间，而不是执行操作。图 8-5 只显示了 Query Cache 的命中率，它与状态变量 `Qcache_hits` 显示的信息相同，Qcache_ 307
hits 是命中的计数器或从缓存中成功检索查询结果的次数。显然，高命中率表明缓存

性能比低命中率时更好，但是反过来却未必正确。低命中率仅仅表示服务器没有使用Query Cache，并不表示服务器性能差。然而，如果想通过使用标准查询提高性能，且发现缓存命中率低，可能是需要优化 Query Cache，或者数据库活动导致多个使缓存结果不可用的事件发生。

Qcache_hits 的值是单调递增的，为了使用这个值测量一段时间内的活动，必须定期重置这个变量。在图 8-5 中，MySQL 管理器实际上是使用 FLUSH STATUS 命令在采样开始时为当前会话将这个变量重置为 0，这是大多数包含采样的监控的常规用法。

图 8-5 中的 Key Efficiency 图动态显示 key buffer 的使用情况。具体而言，这个是 MyISAM key cache，它主要测量多少 key buffer 被使用或多少请求用于读取 key cache。这里使用的状态变量是 key_read_requests。



Key Cache 是 MyISAM 存储引擎用于提高其性能的主要机制，它缓存了经常被频繁访问的索引块（指向数据的指针），以使索引搜索（和随后的数据）可以被更快速地获取。将在第 9 章讨论 Key Cache 的详情。

可以使用图 8-5 来确定何时需要增加 key_cache_size 变量值，以获取更好的性能。也就是说，如果显示的值较高（高百分比），可能需要增加这个值。一般而言，这个值越高，Key Cache 效率越高。相反，如果这个值偏低，说明 MyISAM 表上执行过多的查询，那么此时可能需要减少 Key Cache 的大小。

下面讨论的选项卡将显示更多传统的数值数据。在讨论之前，先介绍如何创建自定义健康图。首先需要右键单击工作区的空白区域，然后选择“New page”，并给这个页面命名，紧接着单击新页面，然后右键单击空白栏选择“New group”，并给这个组命名，最后可以右键单击新组，并为其添加新图。图 8-6 显示了自定义图对话框。

在这里设定需要监控的状态变量和需要执行的计算，并设定数据显示的形式。建议独立
308
完成这步。注意在图 8-6 中我们选择了线性图，并提供了在 ^ 后面括有 [] 的变量名。如果你想要一个更复杂的报告，可以参考公式框后给出的填写提示信息。

还可以设定值的单位、标题选项、最大值和最小值，以及计算最大值的公式。如果为值的标题设定名称，将会在其他图上看到一个小型峰值指示器。图 8-7 显示活动中的图。

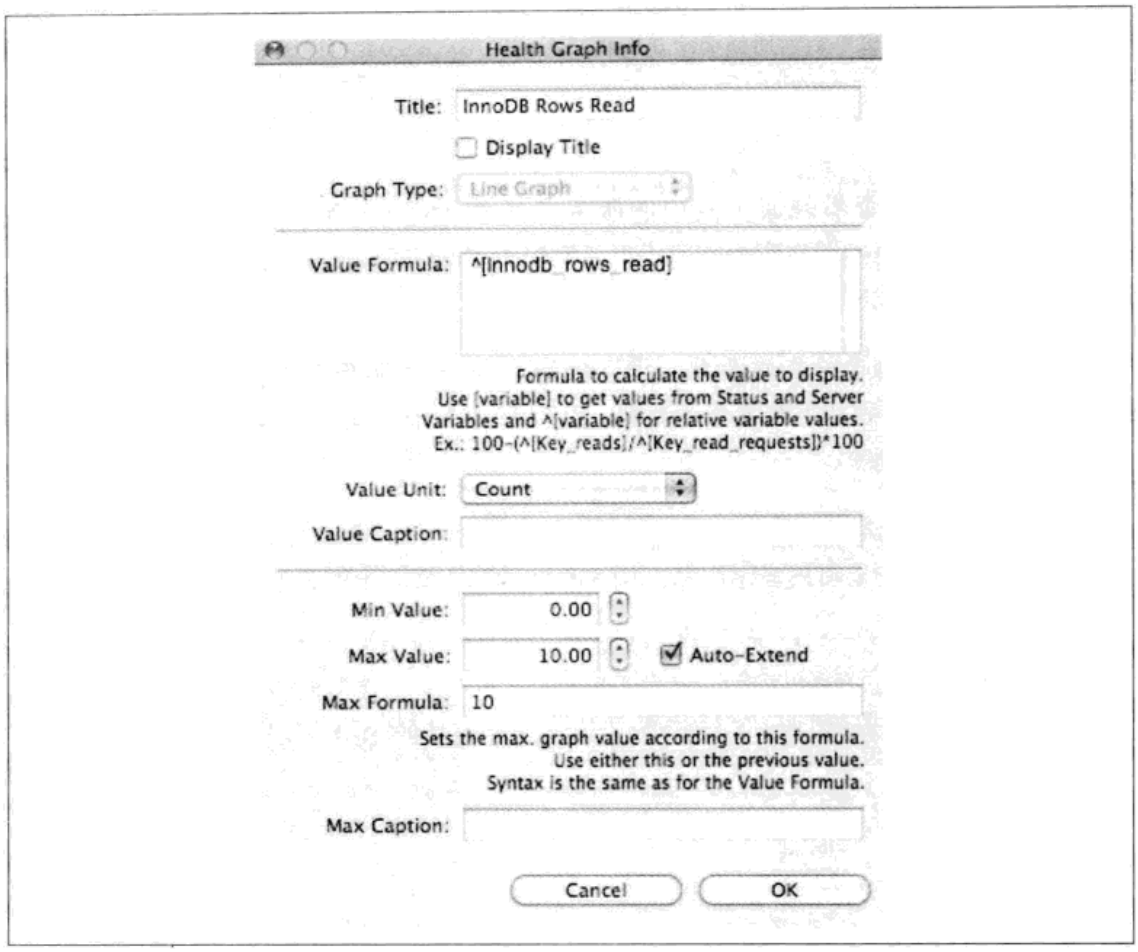


图8-6：自定义图对话框

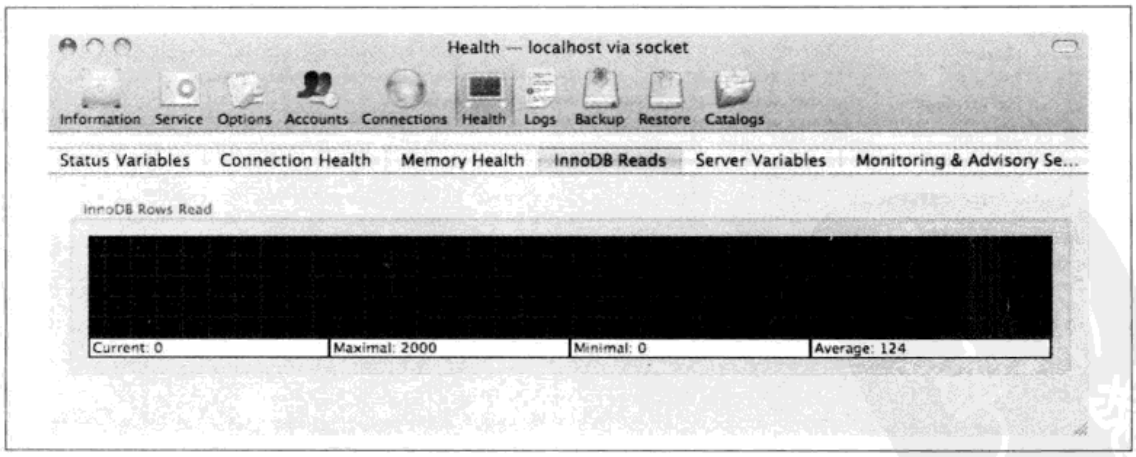


图8-7：自定义图实例



可以右键单击任何标准图并编辑它们。使用这个功能可以查看其他一些图是如何计算的。

如前所述，还可以查看健康工具中的服务器变量（Server Variables）。单击 Server Variables 选项卡，显示图将变为类似于电子数据表的数字图，图左边是个树形控制菜单，这个菜单将所有系统变量分成大类和子类。

中间的面板显示了每个分类的详情。可以通过双击类别名展开这个类别。例如，如果双击 General 类，将看到其下有多个子类别，然后单击 Performance 子类，将看到与性能优化相关的系统变量列表。

虽然这只是性能优化列表的一部分，但是实际上已包含了大部分常用的性能变量。单击其他的项，该组的变量将被显示出来。图 8-8 显示默认安装下的常规性能变量。

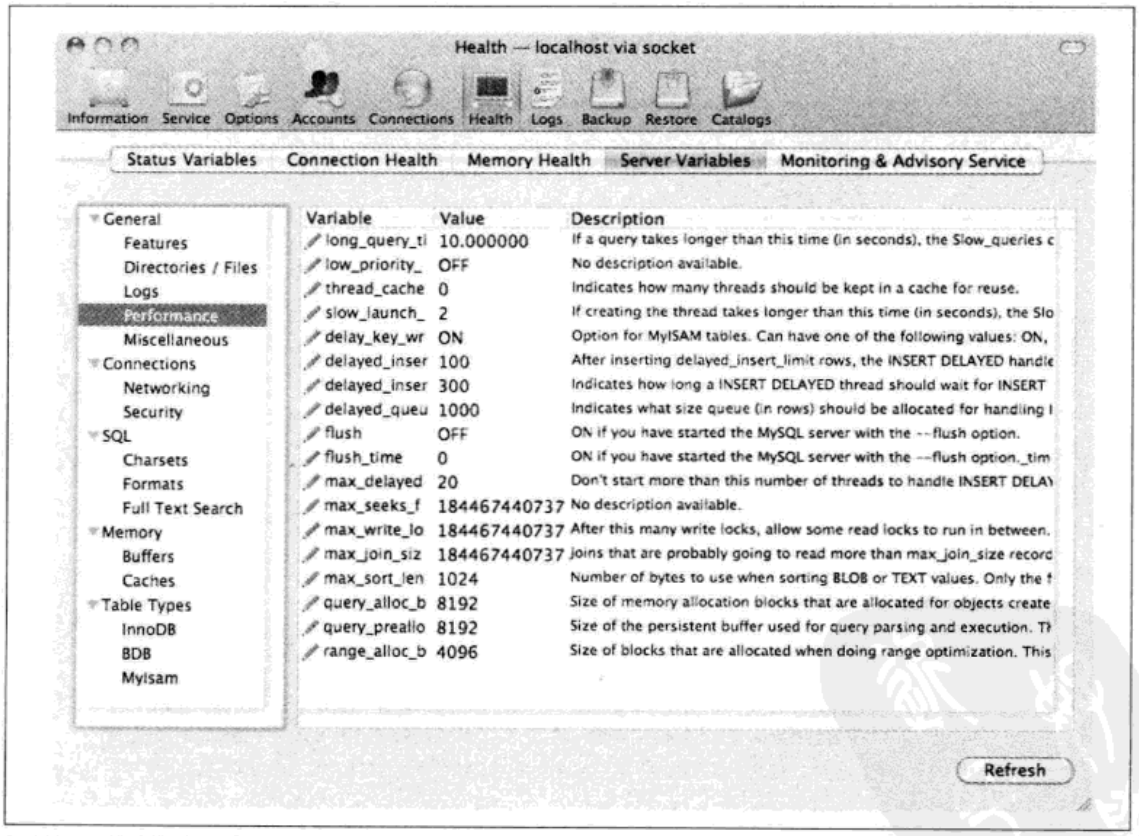


图8-8：服务器变量窗口

这个工具最好的功能是可以动态更改变量值，例如，如果需要增加系统计算慢线程的时

间，只需单击 slow_launch_time 变量的值，然后输入一个新值即可。这个值将在服务器上被自动更改，这个功能使 MySQL 管理器比客户端窗口更易用。



可以单击变量名旁边的铅笔图标来更改这个变量。如果无铅笔图标，表明这个变量是只读的。

当通过 MySQL 管理器更改系统变量，并使用自定义动态健康图查看更改对系统负载的影响时，你正在把 MySQL 管理器当做专业的性能调优工具使用。只需记住调优的黄金准则：一次只更改一个变量，然后反复测试。

Status Variables 选项卡显示了各类系统状态变量。例如，为了查看网络通信方面的所有状态变量，请单击 Networking 大类，然后单击 Traffic 子类。图 8-9 显示了 Status Variables 窗口。

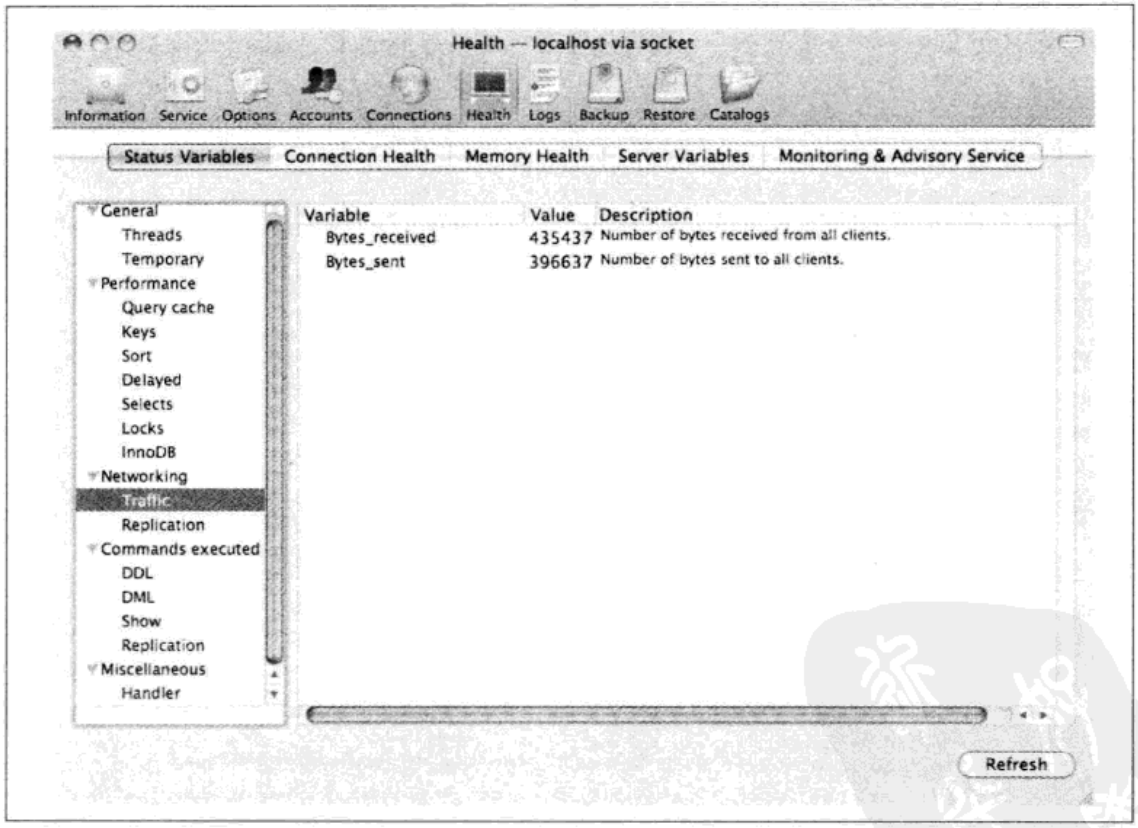


图8-9：Status Variables窗口

在 Server Variables 窗口中，可以查看所有的系统状态变量，这些显示的值是功能组被选

定时产生的值，它们不会自动被更新。单击 Refresh 获取最新的值。与动态健康图不同，刷新过程中返回的值与使用 SHOW STATUS 命令从服务器读取的值相同。在使用这个工具时请记住这一点：它仅显示静态的系统状态变量值，而不显示系统状态变量值的趋势或历史列表。

311 窗口中还包含各个状态变量的简要描述。这对于挖掘有助于诊断性能问题的状态变量非常有用。它还节省了执行 SHOW STATUS 命令和猜测关键字的大量时间（见图 8-5）。

我们要讨论的下一个工具是用于查看服务器日志快照的。我们在后面的“服务器日志”中将详细介绍服务器日志的相关信息，所以在这里只是简单介绍一下服务器日志。

服务器日志记录系统运行时执行的重要事件，其中包含在操作或连接中发生的错误、在服务器上运行的所有查询和运行最慢的查询。

MySQL 管理器允许你挖掘这些日志，虽然它只允许你访问错误的、一般的和慢日志，但是它仍然很有用。

为了查看你所连接的服务器日志，单击工具栏上的 Logs 选项卡，将打开 Logs 工具，如图 8-10 所示。

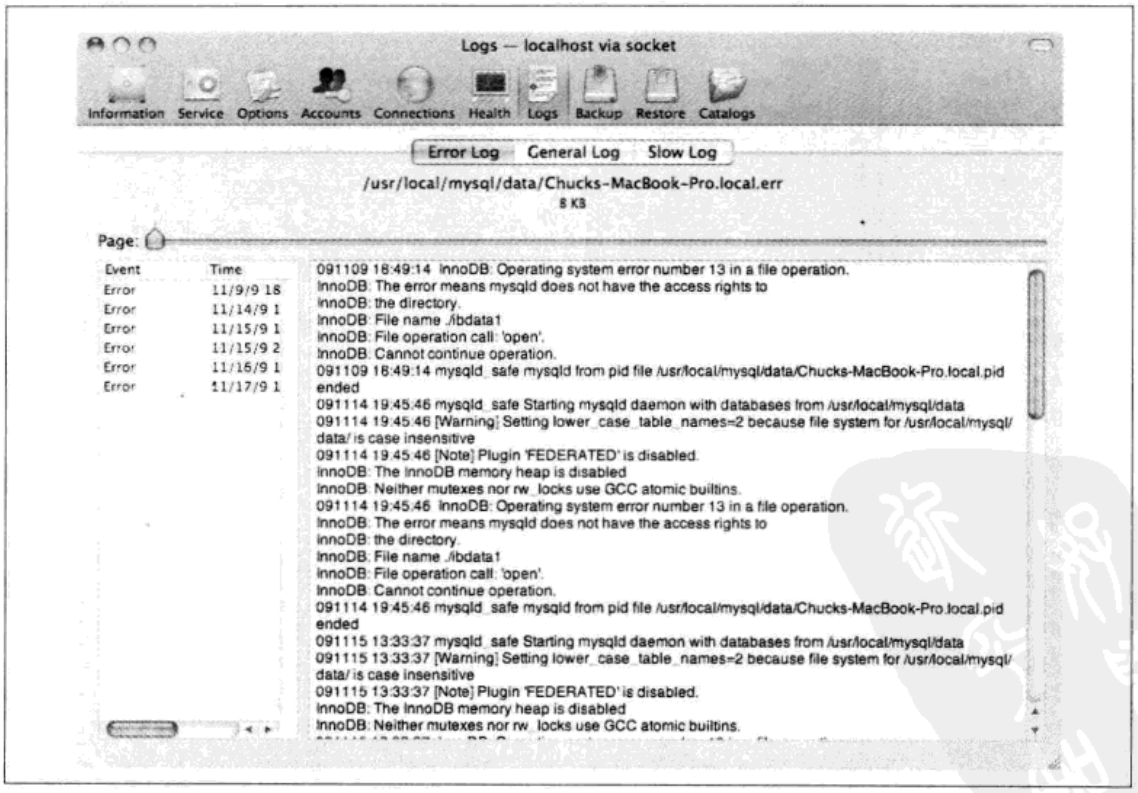


图8-10：Logs窗口



在有些平台上，MySQL 管理器可能需要你输入管理员认证信息才能查看服务器日志或与服务器日志交互。

在整个窗口的顶部，工具栏下面，是一个页面工具，这个工具允许你单击和拖曳工具栏到一个特定页面上，或搜索错误日志页面。

再下面是双窗格两个显示面板，左边面板罗列了日志中的重要事件，右边面板罗列了日志中事件的详情。 312

如你所见，有许多错误被罗列出来了。单击列表中的错误，可以查看其详细信息。当你在整个日志中挖掘特定时间和日期的事件时，这个工具可以节约时间。

现在，你已经看到 MySQL 管理器为高级诊断、性能监控和优化提供了什么，我们希望你在服务器上着手练习高级诊断、性能监控和优化时，考虑使用这个工具。MySQL 管理器删除了大部分使用 SQL 命令检索相同数据的单调功能。另外，MySQL 管理器中的一些新功能，如健康图和自定义健康图，使性能优化变得容易和更及时。

MySQL查询浏览器

MySQL 查询浏览器是 MySQL 的另一个可选的 GUI 工具，使用它可以创建查询并以图形的形式执行这些查询，执行结果集被显示在一个类似于电子数据表的对话框中。MySQL 查询浏览器可以垂直滚动查看所有的结果，并可以调整每列的大小和水平滚动以更方便地查看数据。许多用户发现这个工具比旧的命令行客户端（`mysql`）更方便和容易使用。

为管理员添加的与性能相关的功能和值是 `EXPLAIN` 命令给出的任意给定查询的图形显示。图 8-11 显示从 *sakila* 数据库得到的一个实例，我们将在本章后面详细讨论这些。

这里显示的 MySQL 查询浏览器实例应该给了你一个有效使用 GUI 的说明。可以输入任何查询，并通过首次执行这个查询来查看这个查询的执行计划，然后从 Query 菜单选择 Explain Query。 313

注意这个结果有两部分，底部显示 `EXPLAIN` 命令的结果及实际返回的行，可以使用滚动条查看更多的数据，而不需要重新执行查询。

它之所以是性能调优工具，因为它让你写一次查询，然后使用 `EXPLAIN` 功能，观察其结果，然后重写查询或调整索引，再重新执行查询并观察 GUI 中的变化。或许你以为查询工具只是供用户使用的，但这个工具并非如此。

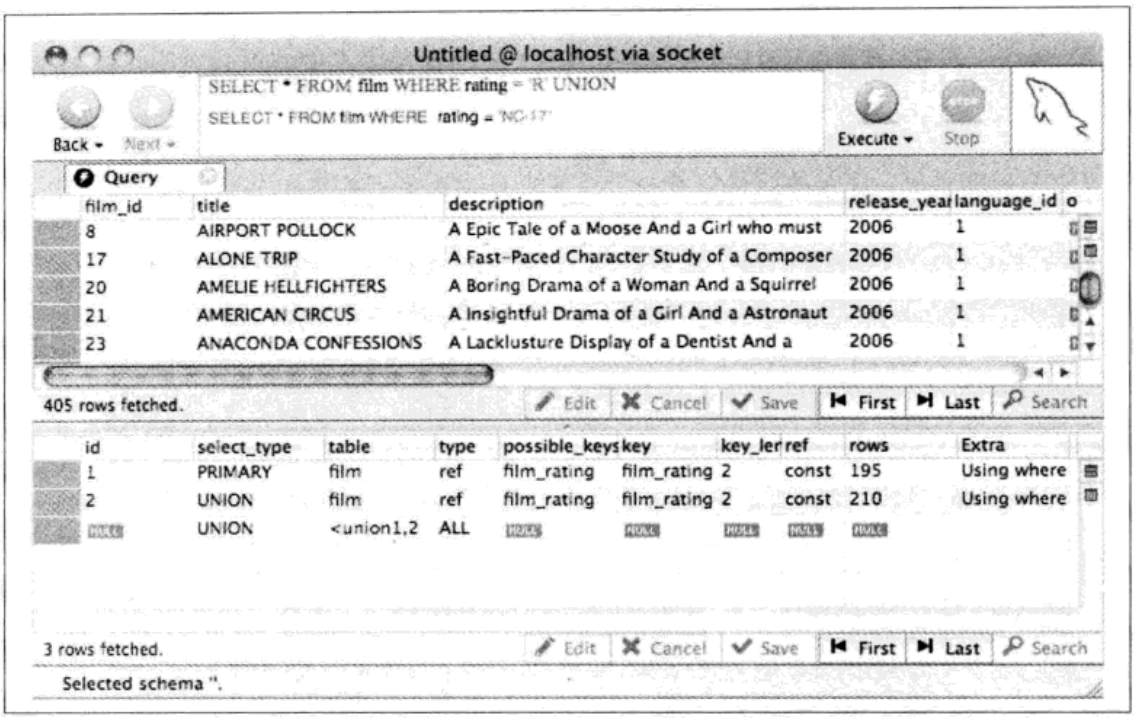


图8-11：MySQL Query Browesr

别急，还有很多。MySQL 查询浏览器增强了编辑工具，如颜色编码，内置于图 8-11 中显示的小文本框中。为了查看 MySQL 查询浏览器的所有高级功能和使用方法，请参看 *http://dev.mysql.com/doc/* 上的 MySQL GUI 工具文档。



如果你运行的是 Windows 系统，可以安装 MySQL System Tray Monitir，它显示了服务器健康一览图。绿色图标表示服务器正在运行，红色图标表示服务器停止。还可以使用这个工具快速访问常用功能，如关闭和启动 MySQL 管理器或 MySQL 查询浏览器。

服务器日志

如果你是一个经验丰富的 Linux 或 UNIX 管理员，那么你应该熟悉日志的概念和重要性。MySQL 服务器在相同的环境中诞生，因此，MySQL 的服务器日志包含关于错误、事件和数据更改的重要信息。

本节探讨 MySQL 服务器中的各种日志，其中包括每种日志在监控和性能改善中的作用。日志文件可以提供大量关于过去事件（实际上是一个时间点发生的事件）的信息。

MySQL 服务器中有几种类型的日志，其中包括记录已经执行过的 SQL 命令的日志、记

录长时间运行（慢）的查询的日志，记录数据更新的日志，以及记录备份和恢复命令的结果的日志。可以使用 startup 选项开启或关闭任意日志。大多数 MySQL 安装程序都至少启用了错误日志，MySQL 服务器中的日志如下所示：

- General query log
- Slow query log
- Error log
- Binary log
- Backup logs

General query log，如它的名称所示，包含有关服务器做了什么的信息，如来自于客户端的连接，以及发送到服务器的命令副本。正如你能想象的，这个日志增长得很快。不论何时试图诊断客户端相关的错误或确定哪个客户端执行某类型的命令，请检查 General query log。



general query log 中的命令的顺序与这些命令从客户端返回的顺序一致，但是这可能不能真实地反映它们的实际执行顺序。

通过设定 `--log` 启动选项开启 General query log，还可以使用 `--log-output` 启动选项指定日志文件的名称，这些选项有等同的动态变量，如 `SET GLOBAL log_output = FILE`，将一个运行中的服务器的日志结果写入一个文件中，最后，可以使用 `SHOW VARIABLES` 命令读取到这些变量的值。

slow query log 保存长时间运行的查询的副本，它与 general log 的格式相同，并且你可以使用与 `--log-slow-queries` 启动选项相同的方式控制 slow query log。服务器变量 `log_query_time`（以秒为单位）用来控制什么查询被记录到 slow query log 中，需要调整这个变量以满足服务器和应用程序的期望，以帮助追踪比期望运行慢的查询。可以单独使用 `FILE`、`TABLE` 将日志条目发送到文件或表中，或同时使用这两个选项将日志条目发送到文件和表中。

slow query log 是个非常有效的工具，它可以在用户抱怨之前追踪出查询问题。当然，最佳状态是保持这个日志很小或者最好一直保持为空，这并不是说你应该将这个变量设置得很高，相反，应该将它设置为你的期望值，并在你的期望值或环境被更改时调整这个变量的值。



Slave 不记录慢查询。然而，如果你使用 --log-slow-slave-statements 选项，它将把运行慢的事件写入 slow log。

error log 包含了 MySQL 服务器启动或停止时收集的信息，还包含服务器运行时产生的错误。当开始分析宕机或受损的 MySQL 服务器时，应该首先查看 error log。在有些操作系统上，error log 还包含了堆栈跟踪（或核心转储）信息。

315 可以使用 `--log-error` 启动选项开启或关闭 error log，error log 的默认名字是主机名附加 .err 后缀，它被保存在基本目录中（与主机的数据目录在相同的位置）。



如果使用 `--console` 启动服务器，错误会被写入标准错误输出结果中，也会被写入 error log 中。

binary log 存储服务器上的数据的所有更改，还记录在服务器上执行的原始命令的统计信息。



MySQL 在线参考手册上这样描述：binary logs 是用来做备份的。然而，实践告诉我们，binarylog 更流行的用途是复制。

binary log 的独特格式允许在增量备份中使用这个日志，可以通过刷新和轮换二进制日志（关闭日志并打开一个新日志），把每次备份时创建的二进制日志文件存储起来，这允许存储自最后一次备份以来所产生的更改。这个相同的技术让你执行 PITR，可以从备份中恢复数据并应用二进制日志到指定点或时间。请参看第 3 章以获取更多关于 binary log 的信息，参看第 12 章以获取更多有关 PITR 的信息。

因为 binary log 记录每个数据更改的副本，所以它给服务器带来一定的负担，但是相对于它的好处而言，这点小代价是值得的。



binary log 的开销可能会很高，这主要取决于磁盘的设置。当 binary log 被启动，使用 InnoDB 存储引擎时不存在同步提交。在二进制日志和 InnoDB 的写频繁的情况下，这一点值得关注。

使用 `--log-bin` 启动项开启 binary log，并为 binary log 设定根文件名。服务器将在文件名的后面附加上一个数字序列，允许自动和手动地执行日志轮换。虽然可以使用 `--log-bin-index` 启动选项更改 binary log 中的索引名，但是一般情况下没有必要这么做。使用 `FLUSH LOGS` 命令执行日志轮换。

还可以分别使用 `--binlog-do-db` 和 `--binlog-ignore-db` 设定日志中记录什么或不记录什么。

第三方工具

有许多第三方工具实际上是非常有用的，其中比较流行的一些工具有 MySAR、mytop、InnoTop 和 MONyog，它们都是基于文本（命令行）的工具，且可以在任意控制台窗口上运行，并通过网络连接任意可获取的 MySQL 服务器。我们将在后面的章节简单讨论这些工具。

MySAR

MySAR 是一个系统活动报告，与 Linux 的 `sar` 命令类似，因此它被称为 MySQL 的 sar 命令。MySAR 收集 `SHOW STATUS`、`SHOW VARIABLES` 和 `SHOW FULL PROCESSLIST` 命令的输出结果，并将这些结果存储在服务器上的 *mysar* 数据库中，并以 *mysar* 命名，且可以使用各种方法配置这个数据集，其中包括限定数据集。另外，为了无限期地继续运行 MySAR，而又不用担心磁盘上填满了 status dumps，可以删除旧数据集。

MySAR 是开源的，它的许可证使用 GNU Public License 第 2 版（GPL v2），可以从 *https://launchpad.net/mysar* 上下载 MySAR。

mytop

mytop 工具监控线程统计信息和 MySQL 的常规性能统计信息，它列出了一些常规统计信息，如主机名称、服务器版本、运行的查询的数量、查询的平均执行时间、线程的总数量和其他重要统计信息，这个工具与 Linux 系统中的 `top` 命令类似。它定期运行 `SHOW PROCESSLIST` 和 `SHOW STATUS` 命令，并像 `top` 命令一样在列表中显示统计信息。Mytop 是由 Jeremy D. Zawodny 开发的，并由他和 MySQL 社区共同维护。图 8-12 显示了 `mytop` 工具的一个例子。

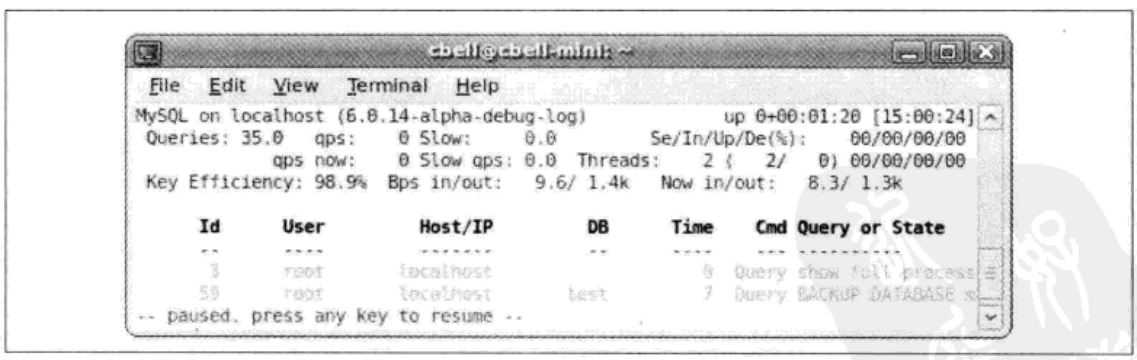


图8-12：mytop实用工具

`mytop` 工具是开源的，且它的许可证使用 GNU Public License 第 2 版（GPL v2），可以在 *http://jeremy.zawodny.com/mysql/mytop/* 上下载 `mytop`。

317

InnoTop

InnoTop 是另一个系统活动报告，类似于 Linux 性能工具，它也与 Linux 的 top 命令相仿，并参考 mytop 工具而设计。虽然它与 mytop 功能相仿，但是它专门用于监控 InnoDB 性能和 MySQL 服务器。主要用于监控事务、死锁、外键、查询活动、复制活动、系统变量的主要统计信息及主机的其他详情。InnoTop 被广泛使用，并被当做常用性能监控工具使用。另外，它拥有许多用于动态监控系统的功能，如果你使用 InnoDB 作为默认存储引擎，并需要一个以文本模式运行的良好的监控工具，请优先选择 InnoTop。图 8-13 显示了 InnoTop 工具的实例。

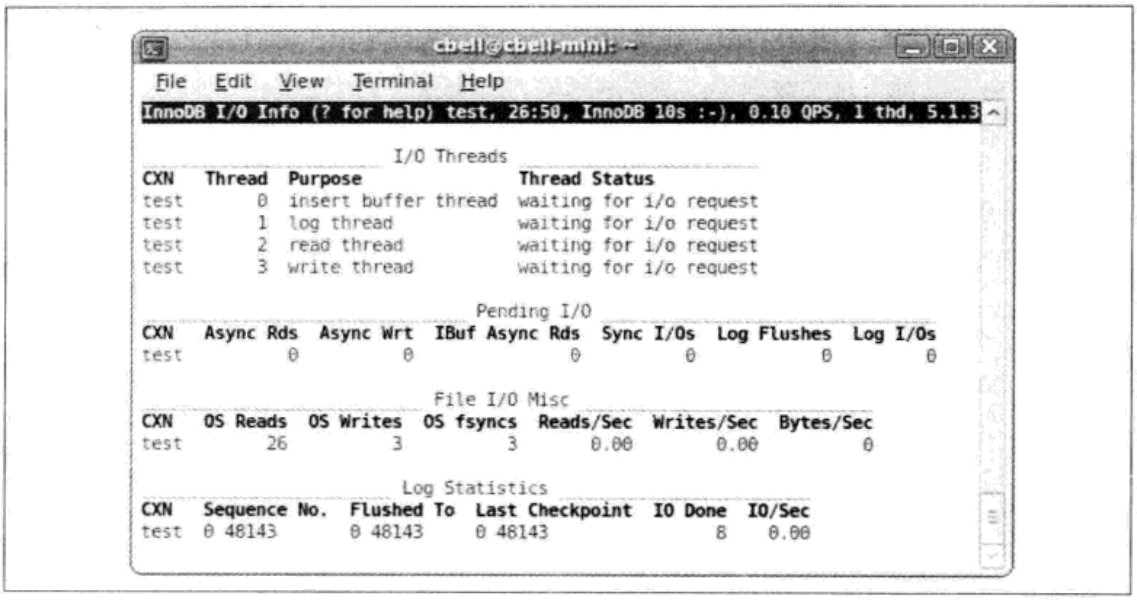


图8-13：InnoTop实用工具

InnoTop 工具的许可证使用 GNU Public License 第 2 版（GPL v2），可以在 *http://innotop.sourceforge.net/* 上下载 InnoTop。

MONyog

MySQL Monitor and Advisor（MONyog）是另一个好的 MySQL 监控工具，它是一个主动监控的解决方案，可以使用它为安全和性能的主要组件设置参数，它还包含有助于服务器性能调优的工具。另外，还可以使用它设置事件以监控特定参数，并在系统达到指定临界点时发出警告。MONyog 的主要功能是：

- 监控服务器资源；
- 识别运行不佳的 SQL 语句；

318

- 监控服务器日志（如 error log）；

- 实时监控查询性能，并识别长时间运行的查询；
- 预警重大事件。

在 *http://www.webyog.com/en/* 上下载 Monyog。

MySQL Benchmark套件

Benchmarking 是这样一个过程：确定系统在某种负载下是如何运行的。Benchmarking 基准各不相同，它还稍微有点艺术形式。Benchmarking 的目的是：在服务器上运行定义好的测试实例，并衡量和记录系统的统计信息，并且这些统计信息记录了服务器轻负载、中等负载和高负载情况下的统计信息。Benchmarking 还可以为系统性能设置期望。

这很重要，因为在系统运行不佳时，它将会给出一些建议。例如，如果在一段时间内用户报告服务器性能差，如何知道服务器运行不佳？假设已经查看了所有常规信息——内存、磁盘等，且所有这些都运行良好，没有出现错误和异常，这时，如果系统运行慢你如何发现？在这种情况下，进入 benchmark，重新运行 benchmark（基准）测试实例，如果执行结果过大（或者过小，具体依赖于你测试什么），将表明系统的性能没有期望的好。

还可以使用 MySQL benchmark 套件建立自己的 benchmark，benchmark 工具被放置在 *sql-bench* 文件夹中，通常被包含在源码发行版中。Benchmark 是用 oPerl 开发的，并使用 Perl DBI 模块访问服务器。如果没有 Perl 或 Perl DBI 模块，请参看 MySQL 在线参考手册中的“在 UNIX 上安装 Perl”章节。

使用以下命令运行 benchmark 套件：

```
./run-all-tests --server=mysql --cmp=mysql --user=root --socket=<socket>
```

这个命令将运行一整套标准 benchmark 测试条目，记录当前结果，并拿当前结果与以前在 MySQL 服务器上运行的结果进行比较。例 8-4 显示了在资源有限的系统上运行这个命令所输出的结果的摘要。

例8-4：MySQL benchmark suite结果

```
cbell@cbell-mini:~/source/bzr/mysql-6.0-review/sql-bench$
Benchmark DBD suite: 2.15
Date of test:        2009-12-01 19:54:19
Running tests on:    Linux 2.6.28-16-generic i686
Arguments:           --socket=../mysql-test/var/tmp/mysqld.1.sock
Comments:
Limits from:         mysql
Server version:      MySQL 6.0.14 alpha debug log
Optimization:        None                                          319
Hardware:
```

```
alter-table: Total time: 77 wallclock secs
  ( 0.12 usr 0.05 sys + 0.00 cusr 0.00 csys = 0.17 CPU)
ATIS: Total time: 150 wallclock secs
  (20.22 usr 0.56 sys + 0.00 cusr 0.00 csys = 20.78 CPU)
big-tables: Total time: 135 wallclock secs
  (45.73 usr 1.16 sys + 0.00 cusr 0.00 csys = 46.89 CPU)
connect: Total time: 1359 wallclock secs
  (200.70 usr 30.51 sys + 0.00 cusr 0.00 csys = 231.21 CPU)
...
```

benchmark 函数

MySQL 包括一个内置函数 benchmark()，可以使用它执行一个简单表达式，并获得一个基准结果，它的最佳用处是测试其他函数或表达式以确定它们是否导致延迟。这个函数有两个参数：循环计数器和需要测试的表达式，以下例子显示了 concat 函数迭代运行 10 000 000 次的结果：

```
mysql> SELECT BENCHMARK(10000000, "SELECT CONCAT( ' te ', ' s ' , ' t ') " ) ;
+-----------------------------------------------------+
| BENCHMARK(10000000, "SELECT CONCAT( 'te' , 's' ,'t')") |
+-----------------------------------------------------+
|                                                   0 +
+-----------------------------------------------------+
1 row in set (0.06 sec)
```

这个函数的值是它运行benchmark函数所花费的时间。在这个例子中，它花费了0.06秒运行迭代。如果你开发一个复杂的查询，请考虑使用这个命令测试它的部分。你会发现问题仅与查询的某个部分有关，而与缺少索引无关。

benchmarking 套件是用于诊断服务器的强大工具。你应该考虑在服务器上运行这个工具来创建性能统计信息的基准，并将这些结果与套件中存储的统计信息进行比较。请查看 MySQL 在线参考手册以获取更多关于 benchmark 套件的信息。

目前我们已经讨论了用于监控 MySQL 的各种工具，并查看了一些最佳实践，接下来，我们将主要关注 MySQL 服务器的更高级功能之一——存储引擎。

数据库性能

监控单个数据库性能是 MySQL 功能设置中为数不多的几个方面之一，MySQL 社区和第三方开发人员已经改善了 MySQL 这些方面的功能。MySQL 包含一些改善性能的基本工具。

但是它们不像其他系统优化工具那样成熟。由于这些限制，大多数 MySQL 管理员通过关系查询优化技术方面的经验获益。我们发现有一些非常棒的参考资料描述了数据库性能的大部分详情，而且有很多读者可能熟悉基础的数据库优化，这里罗列了一些参考资料供大家参考：

320

- *Refactoring SQL Applications*, by Stephane Faroult and Pascal L'Hermite, O' ReillyMedia (*http://oreilly.com/catalog/9780596514976/*)
- *SQL and Relational Theory: How to Write Accurate SQL Code*, by C.J. Date, O'Reilly Media (*http://oreilly.com/catalog/9780596523084/*)
- *SQL Cookbook*, by Anthony Mollinaro, O'Reilly Media (*http://oreilly.com/catalog/9780596009762/*)

我们主要讨论如何使用 MySQL 中的工具优化数据库，而不再重新介绍查询优化技术。下面将使用一个简单的例子和一个已知的数据库样例来说明 MySQL 中查询性能命令的使用，下一节将介绍改善数据库性能的最佳实践。

测量数据库性能

一般的数据库管理系统都提供了分析工具和索引工具，这些报告统计可以用于优化索引。虽然有一些基本元素可以帮助改善 MySQL 数据库性能，但是没有（免费的）高级分析工具可用。

虽然基本的 MySQL 安装包不包含监控数据库改善的正式工具，但是 MySQL 企业版管理套件提供了大量性能监控功能，我们将在第 13 章中更详细地讨论这个工具。

幸运地是，MySQL 提供了一些简单的工具，帮助确定表和查询是否优化。它们都是 SQL 命令，其中包括 EXPLAIN、ANALYZE TABLE 和 OPTIMIZE TABLE。以下章节将详细介绍这些命令。

使用EXPLAIN

EXPLAIN 命令提供关于如何执行 SELECT 语句（EXPLAIN 仅对 SELECT 语句有效）的信息，EXPLAIN 的句法如下所示。请注意 EXPLAIN 与其他数据库系统中的 DESCRIBE 命令类似。

```
[EXPLAIN | DESCRIBE] [EXTENDED] SELECT <select options>
```

还可以使用 EXPLAIN 和 DESCRIBE 命令查看表的列或分区信息，其使用句法如下所示。

```
[EXPLAIN | DESCRIBE] [PARTITIONS SELECT * FROM] <table_name>
```

321



EXPLAIN <*table_name*> 命令与 SHOW COLUMNS FROM <*table_name*> 相同。

我们将讨论 EXPLAIN 命令的第一种用法——检查 SELECT 命令以查看 MySQL 优化程序是如何执行这个语句的。结果包含优化程序预计执行该语句需要的阶梯式的 join 操作列表。



含有 order-by 和 group-by 的查询不是以阶梯格式显示的。

这个命令的最佳用处是确定表是否有正确的索引，并允许更精确的定位候选行。 还可以使用结果测试各种优化程序的重载选项，虽然这是个高级技术，一般情况下不提倡使用，但是在恰当的环境下，你可能遇到这样的查询：和某些优化选项一起运行更快。本节的后面将介绍这样的实例 .

现在，让我们看看 EXPLAIN 命令的一些运行实例，以下实例是在 MySQL 开发和测试时使用的 *sakila* 数据库样例上执行的，请在 *http://downloads.mysql.com/docs/sakila-db.zip* 上下载 *sakila* 样本数据库，*http://forge.mysql.com/wiki/Image:SakilaSampleDB-0.8.png* 上可以找到 MySQL vForge 的 E-R 图。



我们将使用 \G 或垂直显示格式以获取更清晰的显示结果。

让我们从一个简单且看起来没有什么危害的查询开始，比方说，希望看到所有比 PG 等级高的影片，结果集包含以下列的单条信息：

- `id`
 语句的执行序列号。
- `select_type`
 被执行的语句的类型。

- `table`
 在这一步操作的表。
- `type`
 使用的 join 类型。



如果这个字段显示 ALL，则执行全表扫描，应该通过添加索引或重写查询的方式来避免这些操作。类似地，如果这个字段显示 INDEX，则执行全索引扫描，这是非常低效的。参考 MySQL 在线参考手册可以获取更多 join 类型和它们的意义。

- possible_keys
 可用字段的列表，如果索引可用。
- key
 被优化程序选中的主键。
- key_len
 主键或主键被使用部分的长度。
- ref
 约束或进行比较的字段。
- rows
 估计要处理的行数。
- extra
 优化程序的额外信息。

例 8-5 显示了 MySQL 优化程序如何执行这个语句。

例8-5：一个简单的SELECT语句

```
mysql> EXPLAIN SELECT * FROM film WHERE rating >  'PG' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 892
        Extra: Using where
1 row in set (0.01 sec)
```

从这个输出结果可以看到优化程序只执行了一步，而且没有使用任何索引。这是明智的，因为没有使用任何索引列。此外，即使这里包含了一个 WHERE 子句，但是优化程序仍然需要做全表扫描。当你考虑使用的列和缺少索引时，这可能是正确的选择。然而，如果 323 这个查询运行几百上千次，全表扫描将很浪费时间。在这种情况下，从结果可以看出添加索引可以改善执行情况。给表添加一个索引，然后再测试。例 8-6 显示了改善查询的计划。

例8-6：改善的查询计划

```
mysql> ALTER TABLE film ADD INDEX film_rating (rating);
Query OK, 0 rows affected (0.42 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * FROM film WHERE rating >  'PG' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: film_rating
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 892
        Extra: Using where
1 row in set (0.00 sec)
```



那些已经发现问题的人请忍一下，因为我们将会解决这个问题。

在这里，我们看到该查询已经识别了一个索引（pissible_keys），但是仍然没有使用索引，因为 key 字段是 NULL，那么我们可以做些什么呢？对于这个简单的例子，你可能会注意到只有 892 行预计将被读取，而实际被读取的行有 1 000 行，而且结果集中将只包含 418 行。显然，如果这个查询只读取所有行的 42%，它将执行得更快。

现在看看通过使用 EXTENDED 关键字是否可以从优化程序中获得一些额外的信息。这个关键字可以通过 SHOW WARNINGS 命令显示一些额外的信息，应该在 EXPLAIN 命令后面立即执行这个命令。这个警告文本描述了优化程序是如何识别语句中的表和列名、如何在内部重写这个查询、如何应用各个优化规则的，还描述了有关执行的其他信息说明。例 8-7 显示了使用 EXTENDED 关键字的结果。

324

例8-7：使用EXTENDED关键字获取更多信息

```
Mysql> EXPLAIN EXTENDED SELECT * FROM film WHERE rating >  'PG' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: film_rating
          key: NULL
      key_len: NULL
```

```
          ref: NULL
         rows: 892
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS \G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select `sakila`.`film`.`film_id` AS `film_id`,
`sakila`.`film`.`title` AS `title`,`sakila`.`film`.`description` AS `description`,
`sakila`.`film`.`release_year` AS `release_year`,
`sakila`.`film`.`language_id` AS `language_id`,
`sakila`.`film`.`original_language_id` AS `original_language_id`,
`sakila`.`film`.`rental_duration` AS `rental_duration`,
`sakila`.`film`.`rental_rate` AS `rental_rate`,
`sakila`.`film`.`length` AS `length`,
`sakila`.`film`.`replacement_cost` AS `replacement_cost`,
`sakila`.`film`.`rating` AS `rating`,
`sakila`.`film`.`special_features` AS `special_features`,
`sakila`.`film`.`last_update` AS `last_update`
from `sakila`.`film` where (`sakila`.`film`.`rating` > 'PG')
1 row in set (0.00 sec)
```

这次有一个警告（用于包含来自于优化程序的信息）：查询的重写形式包含了所有的字段，并显式引用了 WHERE 从句中的列。不幸的是，这个简单的查询需要考虑更多以使它运行更有效。你可能会遇到这样的查询：强迫你重新设计查询、重新考虑频繁使用它，或者（更可能）考虑重新设计表以支持更好的索引。

当执行一个特定 rating 的查询而不是使用范围查询时，让我们看看发生了什么。我们将看到使用和不使用索引的优化。例 8-8 显示了结果。

例8-8：删除范围查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ref
possible_keys: film_rating
          key: film_rating
      key_len: 2
          ref: const
         rows: 195
        Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> ALTER TABLE film DROP INDEX film_rating;Query OK, 0 rows affected (0.37 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 892
        Extra: Using where
1 row in set (0.00 sec)
```

现在看到有一些小小的改善，注意第一个查询计划确实使用了索引，而且是一个不使用索引的大改善计划。疑问依然存在，为什么优化程序不使用索引？在这种情况下，我们在枚举字段上使用了非唯一索引，这听起来像是个好办法，但是实际对于枚举值的范围查询却没有多大帮助。然而，我们可以重写上面的查询（实际上使用几个不同的方法），以获得更好的性能。让我们再来看看这个查询。

我们希望所有影片的 rating 比 PG-13 更高，假定对 rating 进行了排序，且枚举字段反映了顺序，在 sakila 数据库中，rating 字段被定义为这几个值：G、PG、PG-13、R、NC-17。因此，如果枚举索引的每个值与这个顺序（如 G=1，PG=2 等）对应，那么它看起来是维持了原有的顺序，但是如果这个顺序不正确或（像这个例子一样）这个表的值不完整呢？

在我们选择的这个例子中，我们希望所有影片的 rating 比 PG-13 高，从 rating 列表中我们看到有些影片的 rating 是 R 或 NC-17。在不使用范围查询的前提下，如果列举了所有的这些值，让我们看看优化程序做了些什么。

回想一下，因为删除了索引，所以我们首先试试没有索引的查询，然后添加索引，再看看查询是否有所改善。例 8-9 显示了改进后的查询。

326

例8-9：改进后的非范围查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' OR rating = 'NC-17' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
```

```
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 892
        Extra: Using where
1 row in set (0.00 sec)

mysql> ALTER TABLE film ADD INDEX film_rating (rating);
Query OK, 0 rows affected (0.40 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' OR rating = 'NC-17' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: film_rating
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 892
        Extra: Using where
1 row in set (0.00 sec)
```

可是还不行，我们同样选择了查询有索引的列，但是优化程序却不能使用这个索引。我们知道，优化程序将为一个简单等值对比使用索引。重写这个查询，使用 union 连接两个查询。例 8-10 显示了重写后的查询。

例8-10：使用UNION重写的查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' UNION
SELECT * FROM film WHERE rating = 'NC-17' \G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: film
         type: ref
possible_keys: film_rating
          key: film_rating
      key_len: 2
          ref: const
         rows: 195
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: UNION
        table: film
```

```
         type: ref
possible_keys: film_rating
          key: film_rating
      key_len: 2
          ref: const
         rows: 210
        Extra: Using where
*************************** 3. row ***************************
           id: NULL
  select_type: UNION RESULT
        table: &lt;union1,2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra:
3 rows in set (0.00 sec)
```

这次运行成功了，可以看到这个查询计划使用了索引并处理更少的行。从 EXPLAIN 命令的结果中可以看到优化程序单独运行每个查询（从第一步到第 *n* 步一步步执行），并在最后一步合并结果。



MySQL 有一个会话状态变量 last_query_cost，这个变量存储最后一次执行的查询消耗的时间。使用这个变量比较同一个查询的两个查询计划，例如，执行完每个 EXPLAIN 后，查看这个变量的值，拥有最低成本值的查询被认为是更高效（消耗更少的事件）的查询。如果该值为 0，则表明没有查询被编译提交。

虽然这个试验看起来可能需要做很多工具却收获很小，但是应该这样考虑：应用程序中有很多这样的查询正在执行，而没有人注意到它的低效。通常只有在行数大到足够引起注意的时候才会遇到这样类型的查询。在 *sakila* 数据库中，只有 1 000 行，但是如果有一百万或数千万行呢？

除了 EXPLAIN 以外，标准 MySQL 发行版中再也没有单一的工具可用于剖析查询。MySQL 在线文档中的“优化”这一章节介绍了大量技巧和窍门来帮助有经验的 DBA 提高各种形式的查询的性能。

328

使用ANALYZE TABLE

与大多数传统的优化程序相同，MySQL 优化程序使用表的统计信息进行最优查询执行计划的分析，这些统计信息涉及多个项目的信息，其中包括索引、数值分布和表结构。

ANALYZE TABLE 命令重新计算一个或多个表的主键分布。这个信息确定一个 join 操作中

表的顺序。ANALYZE TABLE 命令的使用语法如下所示：

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE <table_list>
```

你可以为 MyISAM 和 InnoDB 表更新主键分布。这一点值得注意，因为它不是适用于所有存储引擎的普通工具。但是，如果存储引擎支持索引，那么所有的存储引擎必须向优化程序报告索引基数的统计信息。有些存储引擎，如第三方引擎，有它们特定的内置的统计信息。例 8-11 显示了这个命令的典型执行情况，在没有索引的表上运行这个命令是无效的，但是也不会导致错误发生。

例8-11：分析表并更新主键分布

```
mysql> ANALYZE TABLE film;
+------------+---------+----------+----------+
| Table      | Op      | Msg_type | Msg_text |
+------------+---------+----------+----------+
| sakila.film  | analyze | status   | OK       |
+------------+---------+----------+----------+
1 row in set (0.00 sec)
```

在这个例子中，我们看到分析完成了，并且没有异常出现。在这个命令执行的过程中如果出现异常事件，Msg_type 字段将会显示“info”“error, 或“warning”，在这些情况下，Msg_text 字段将显示事件的额外信息，如果你得到的结果不是“status”和“OK”，一般来说应该调查一下。

可以使用 SHOW INDEX 命令查看索引的状态，例 8-12 显示了 film 表的输出结果样本。在这个例子中，我们对每个索引的基数感兴趣，它是索引中的唯一值的数量估计。为方便起见我们在显示结果中省略了其他列。请参看 MySQL 在线参考手册以获取更多关于 SHOW INDEX 的信息。

例8-12：film表的索引

```
mysql> SHOW INDEX FROM film \G
*************************** 1. row ***************************
       Table: film
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: film_id
   Collation: A
 Cardinality: 1028
...
*************************** 2. row ***************************
       Table: film
  Non_unique: 1
    Key_name: idx_title
Seq_in_index: 1
```

```
 Column_name: title
   Collation: A
 Cardinality: 1028
...
*************************** 3. row ***************************
       Table: film
  Non_unique: 1
    Key_name: idx_fk_language_id
Seq_in_index: 1
 Column_name: language_id
   Collation: A
 Cardinality: 2
...
*************************** 4. row ***************************
       Table: film
  Non_unique: 1
    Key_name: idx_fk_original_language_id
Seq_in_index: 1
 Column_name: original_language_id
   Collation: A
 Cardinality: 2
...
*************************** 5. row ***************************
       Table: film
  Non_unique: 1
    Key_name: film_rating
Seq_in_index: 1
 Column_name: rating
   Collation: A
 Cardinality: 11
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
5 rows in set (0.00 sec)
```

LOCAL 或 NO_WRITE_TO_BINLOG 关键字可以防止命令被写入二进制日志（从而在复制拓扑中被复制）。如果希望在复制数据时进行实验或调整，或者如果希望从二进制日志中省略这一步，且在 PITR 中不再重复之，那么这个命令将对你很有帮助。

无论表何时发生重大更新（例如批量加载数据），都应该执行这个命令，在这个操作过程中，系统必须在这个表上设定一个读锁。

330

使用OPTIMIZE TABLE

被频繁更新和删除的表将很快会变得支离破碎，并取决于存储引擎的不同，将会出现不同程度的闲置空间或不最理想的存储结构。支离破碎的表会导致性能下降，尤其是在表

扫描过程中。

OPTIMIZE TABLE 命令可以重构一个或多个表的数据结构。对于长度可变的行而言，这个命令尤其有用，OPTIMIZE TABLE 命令的句法如下：

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE <table_list>
```

可以在 MyISAM 和 InnoDB 表上使用这个命令，这一点非常值得注意，因为它并不适用于所有的存储引擎。如果表不能被重新组织（例如如果不存在可变长度的记录或不存在碎片），这个命令将恢复重建表，并更新统计信息，例 8-13 显示了这个命令的一个实例。

例8-13：optimize table命令

```
mysql> OPTIMIZE TABLE film \G
*************************** 1. row ***************************
   Table: sakila.film
      Op: optimize
Msg_type: note
Msg_text: Table does not support optimize, doing recreate + analyze instead
*************************** 2. row ***************************
   Table: sakila.film
      Op: optimize
Msg_type: status
Msg_text: OK
2 rows in set (0.44 sec)
```

我们在这个结果集中看到两行信息，第一行表明 OPTIMIZE TABLE 命令不能被执行，且这个命令将被 re-create 和 ANALYZE TABLE 命令代替。第二行是显示 ANALYZE TABLE 命令运行的结果。

ANALYZE TABLE 命令执行过程中发生的任意异常事件都会在 Msg_type 字段中被表现出来，在这种情况下，Msg_type 字段会显示以下信息："info""error" 或 "warning"，另外，Msg_text 字段将会显示事件的额外信息。如果 Msg_type 和 Msg_text 字段显示的不是 "status" 和 "OK"，一般来说应该调查一下。

LOCAL 或 NO_WRITE_TO_BINLOG 关键字防止命令被写入二进制日志中（因此它将不会被复制）。如果希望在复制数据时进行实验或调整，或者如果希望从二进制日志中省略这一步，且在 PITR 中不再重复，那么这个命令将对你很有帮助。

无论表何时发生重大更新（例如大量删除和插入），都应该执行这个命令。这个操作主要用于优化数据元素排列结构，而且它比预期运行的时间更长，这个操作适合在低负载时运行。 331



在使用 InnoDB 时，尤其当有辅助索引（通常会导致碎片产生）时，你可能没有看到任何改进，或者你可能遇到这个操作运行时间很长，除非使用 InnoDB“快速索引创建”选项。

数据库优化的最佳实践

如前所述，有许多很好的例子、技术和方法被世界上最好的数据库性能专家所高荐。我们将讨论提高数据库性能的最常用的方法，而不是评论或建议任何特定的工具或技术。建议读者参看前面提到的文本参考资料以获取关于这些方法的更多详情。

谨慎而有效地使用索引

大多数数据库专家知道索引的重要性以及它们是如何提高性能的。使用 EXPLAIN 命令是确定需要哪些索引的最好办法。虽然索引不足引起的问题是可以理解的，但是有时候拥有太多的索引也会引起性能问题。

正如你在使用 EXPLAIN 命令时看到的，有可能创建很多的索引、没有索引或索引没有被使用。索引会在对表进行删除和插入时增加开销。在有些情况下，索引过多且分布很广时，会大大降低插入和删除的性能。它还会降低复制和恢复操作的性能。

应该定期检查索引以确保它们都是有意义的且被使用了。应该删除以下索引：没有被使用、使用有限制或分布很广的索引。通常可以使用规范化解决一些分布广的问题。

使用规范化，但不要使用过头

许多学习计算机科学或相关学科的数据库专家可能对 C.J. Date 和其他人描述的规范形式有美好的（或噩梦式）的回忆。我们不再在这里重温这些知识，而是讨论这些经验教训的影响。

规范化（至少是第三范式）是一个易于理解且标准的方法。然而，在有些情况下，你可能希望违反这些规则。

查找表的使用通常是规范化的产物，也就是说，你创建了一个特殊的表，这个表包含了
332 在其他表中被频繁使用的相关信息的列表。然而，当使用那些经常被访问且分布有限（仅有或有限的行数拥有小值）的查找表时，会使系统性能降低。在这种情况下，每次你使用查询信息，它们必须使用 join 以获取完整数据。join 的开销很大，而且频繁访问会使开销随着时间逐渐增加。为了减少这种潜在的性能问题，可以使用枚举字段存储数据，而不是使用查找表存储数据。例如，可以使用枚举字段存储头发颜色值，而不是创建表来存储头发颜色值（尽管有些亚文化可能坚持要求这样，但是头发的颜色类型真的很有限），这样还可以避免使用 join。

另一个潜在问题是关于计算字段的。通常情况下，我们不存储由其他数据形成的数据（如销售费或几个列的总和）。相反，计算数据要么在数据检索过程中通过视图执行，或者在应用程序中执行。如果计算很简单，或者很少被执行，这可能不是一个真正的问题，但是如果计算很复杂且被频繁执行呢？在这种情况下，可能在执行这些计算上会浪费很多时间。缓解这个问题的办法是使用触发器计算值，并将结果存储在表中。虽然从技术上而言会复杂化数据（规范化的一大忌），但是它可以在有大量计算执行时提高性能。

使用正确的存储引擎

MySQL 的最强大的功能之一是它支持不同的存储引擎，存储引擎管理如何存储和恢复数据。MySQL 支持多个存储引擎，每个存储引擎都具有独特的功能和用途，可以使数据库设计者通过使用最适合他们的应用程序的存储引擎来改善数据库系统的性能。例如，如果有一个这样的环境：使用事务控制高度活跃的数据库，请选择一个适合这个情况的存储引擎（是的，Virginia，MySQL 中的有些存储引擎不支持事务），你还可能会发现这样的视图或表，它们常常被查询但是几乎不被更新（例如查找表），在这种情况下，你可能希望使用存储引擎将这些数据存储在内存中，以便快速访问它们。

最新的 MySQL 版本中的有些存储引擎是以插件形式开发的，有些发行版的 MySQL 在默认情况下仅启用某些存储引擎。为了发现哪些存储引擎被启用，请执行 SHOW ENGINES 命令查看，例 8-14 显示了一个典型 MySQL 安装程序中的存储引擎。

例8-14：*存储引擎*

```
mysql> SHOW ENGINES \G
*************************** 1. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 2. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
```

```
  Savepoints: NO
*************************** 4. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
8 rows in set (0.00 sec)
```

334 这个结果集包含了以下信息：所有已知的存储引擎、存储引擎是否被安装和配置（Support=YES）、存储引擎的功能和存储引擎是否支持事务、分布式事务（XA）或保存点。



保存点是一个指定的事务，可以像使用事务一样使用它。可以建立一个保存点，并释放（删除保存点）或从保存点开始回滚更改。参看 MySQL 在线参考文档可以获取更多关于保存点的信息。

由于有这么多的存储引擎供选择，所以在为了提高性能选择存储引擎时可能有些困惑。下面简单介绍每个存储引擎，其中包括它们的最佳使用方法。可以在 CREATE 语句中使

用 ENGINE 参数为表选择存储引擎，另外，可以使用 ALTER TABLE 命令更改存储引擎。

```
CREATE TABLE t1 (a int) ENGINE=InnoDB;
ALTER TABLE t1 ENGINE=MEMORY;
```

InnoDB 存储引擎支持事务，在需要事务支持时，通常应该选择这个存储引擎，它是 MySQL 中目前唯一事务性的引擎。有很多第三方存储引擎支持事务，但是仅有 InnoDB 有“开包即用”选项。有趣的是，InnoDB 中所有的索引都是 *B-trees*，在这个 B 树中索引记录被存储在树的叶子页，InnoDB 适用于高性能和事务处理环境。

MyISAM 存储引擎是 MySQL 的默认引擎，如果你在 CREATE 语句中省略了 ENGINE 选项，那么默认使用这个引擎。MyISAM 经常在数据仓库、电子商务和企业应用中使用。MyISAM 使用高级缓存和索引机制提高数据检索和索引速度，另外，当各种应用程序需要快速检索数据而不需要事务时，MyISAM 将是很好的选择。

Blackhole 存储引擎是非常有趣的，它并不存储任何东西。实际上，正如它的名字所言——存储进去的数据永远不会返回。Blackhole 存储引擎有个特殊的用途，如果启用了二进制日志，SQL 命令将被写入这个日志，这时，Blackhole 存储引擎被当做复制拓扑中的中继代理使用，在这种情况下，中继代理执行来自于 Master 的数据，并将这些数据发送到它的 Slave 上去，但是它本身并不存储任何数据。当你希望测试应用程序以确保它写入数据、但是并不希望在磁盘上存储任何数据时，Blackhole 存储引擎比较有用。

CSV 存储引擎可以创建、读取和写入逗号分隔值（CSV）文本。CSV 存储引擎的最佳用处是将结构业务数据快速导成电子数据表，它不提供任何索引机制，且在存储和转换日期 / 时间值（在查询中它们不遵循本地时间）上存在某些问题。另外，当你想允许其他 335 应用程序以共同的格式共享或交换数据时，CSV 存储引擎的用处很大。鉴于它存储数据效率不高，应该谨慎使用。



CVS 存储引擎用于写日志文件，如备份日志是 CSV 文件，可以被使用 CSV 协议的应用程序打开（但是不能在服务器运行时打开）。

Memory 存储引擎（有时被称为 HEAP）是内存中的存储器，它使用哈希机制频繁检索被使用过的数据，这样可以更快地检索，它访问数据的方式与其他存储引擎类似，但是数据存储在内存中，并且只在 MySQL 会话中有效。当关机时，这些数据被刷新并删除掉。Memory 存储引擎通常用于以下情况：静态数据被频繁使用且很少被改变（如查找表）。例子中包含的 zip code 列表、state 和 country namespace、category 列表及其他数据被频繁访问，且几乎不被更新。还可以在这样的数据库中使用 Memory 存储引擎：利用快照技术访问分布的或历史数据。

Federated 存储引擎创建引用多个数据库系统的单个表，它允许你将跨数据库服务器的表连接到一起。这个机制有点类似于在其他数据库系统中可获得的链接数据表。Federated 存储引擎最适用于分布式或数据集环境，它的最有趣的特性是：不移动数据，也不要求远程表去使用相同的存储引擎。



Federated 存储引擎目前在 MySQL 的大部分发行版中不可用，请参看 MySQL 参考手册以获取更多关于它的详情。

Archive 存储引擎可以存储大量压缩数据，它最适用于存储和检索大量很少访问的存档或历史数据，但是它不支持索引，且只能通过表扫描被访问。因此，不应该在平常的数据库存储和检索中使用 Archive 存储引擎。

Merge(MRG_MYISAM) 存储引擎可以使用相同的结构封装一组 MyISAM 表，并把这些表的集合看成一个单表。因此，这些表按单个表的位置分区，但是没有使用额外的划分机制。所有的表都必须放在统一的服务器上（但不需要在同一个数据库中）。

336



当在合并表上执行 DROP 命令时，只有 Merge 规范被删除，原始表并没有被改变。

Merge 存储引擎的最佳特性是速度，它允许将一个大表分割成许多不同的小块，并存储在不同的磁盘上。可以使用 merge 表把这些小块合并，并且可以同时访问这些小块。搜索和排序将执行得更快，因为每个小块中需要执行的数据较少。另外，表的修复也更高效，因为修复多个更小且独立的表比修复一张大表更快、更容易。不幸的是，这种配置有几个缺点：

- 必须使用相同的 MyISAM 表组成一个合成表。
- 替换操作不可用。
- 索引比单表的索引效率低。

Merge 存储引擎最适合于非常大的数据库（VLDB）应用程序，如数据仓库，其中数据存储在多个数据库的多个表中。还可以使用它帮助解决数据划分问题：你想水平划分数据，但是不希望增加划分表选项的复杂性。

显然，由于供选择的存储引擎有很多，有可能所选择的存储引擎会使性能下降，或者在某些情况下，禁止某些解决方案。例如，如果你在创建表时从不指定存储引擎，MySQL 使用默认的存储引擎。如果没有手动设置，默认存储引擎将恢复为平台特定的默认存储

引擎，有些平台上使用的是 MyISAM。这可能意味着由于不支持事务，你可能错过了优化查找表或限制了你的应用程序功能。在设计或优化数据库时，最好额外花点时间分析一下存储引擎的选择。

通过Query Cache使用视图来加速结果

视图是封装复杂查询的一个简便方法，并简化了数据处理工作。可以使用视图来水平地（更少的列）或垂直地（SELECT 语句中使用 WHERE 子句）限制数据，两者都很方便。当然，更复杂的视图使用这两种限定返回结果集给用户，或者隐藏某些基表，或确保执行有效的 join。

使用视图来限定返回列可以在某些你没考虑到的地方帮助你，它不仅降低了数据的处理量，还可以帮助你避免用户在无意中执行的 SELECT* 操作所带来的高额代价。当大量这类型的操作运行时，你的应用程序执行了过多的数据，这不但影响应用程序的性能，而且影响服务器的性能，更重要的是，它会降低可用网络带宽。通常最好使用视图限定数据，并隐藏对基表的访问，这样避免了试图直接访问基础表的用户。

337

限定结果集返回的行数的视图还可以减少网络带宽的使用量，并可以提高应用程序的性能，这些类型的视图还可以避免使用 SELECT* 查询。以这种方式使用视图还需要多一点规划，因为你的目的是创建有意义的数据子集。需要了解数据库的需求，并了解查询使用正确的 WHERE 子句。

只需一点点努力，你就可能发现你可以创建既有水平限定又有垂直限定的视图，从而确保应用程序仅处理那些需要的数据。移动的数据越少，在相同的时间内，应用程序可以处理的数据越多。

也许视图的最佳用处是减少运行不佳的 join 操作，尤其是当有一个复杂的规范化模式时，对于用户而言，可能并不清楚如何合并表以形成有意义的结果集。实际上，为了获取更好的性能，DBA 所做的大部分工作都是与运行不佳的 join 操作相关的，有时，这样做的效果可能很小，比如，join 操作中只执行了很少的行，但是大部分情况下，改善响应时间很重要。

在 MySQL 中使用 Query Cache 时视图也起很大作用，Query Cache 存储频繁使用（访问）的查询的结果，使用提供标准结果集的视图可以提高结果被缓存的概率，从而使检索更高效。

可以通过一些设计工作和视图的使用来提高性能，花时间去研究被迁移的数据量（列的数量和行的数量），并检查应用中使用 join 的查询，花些时间创建限定数据的视图，识别最有效的 join 操作，并把它们封装在视图中，想象一下，你会更容易放过你的用户正

在执行的有效的 join。

使用约束

约束提供了另一个改善性能问题的工具。这里且不讨论使用约束的限制，鼓励使用约束的标准使用方法，而不赞成采取事后补救的策略。

MySQL 上有很多种类型的约束可用，如下所示：

- 唯一索引；
- 主键；
- 外键；
- 枚举值；
- 集合；
- 默认值；
- NOT NULL 选项。

338

我们已经讨论过如何使用索引和过度使用索引，索引有助于提高数据检索速度，使得系统更快地存储和查找数据。

唯一索引是表中某个字段上的简单索引，并保证在表中这个字段不使用 NOT NULL 约束的前提下该字段没有重复的值。也就是说，索引中每行的值不同。在没有重复值的字段上使用唯一索引，如序列号、订单号、社会安全码等。一个表可以有多个唯一索引。

主键也可以被看成唯一索引，但是它是表中每行的唯一标识，并禁止重复主键的创建，当设计有效时，可以在表连接时形成 join 列。

最常用的主键是自增序列，唯一标识一行，MySQL 提供了用于系统自动创建唯一值的 AUTO_INCREMENT 选项。代理主键值的使用被有些数据库理论看成是一种妥协，有些不建议使用它，因为主键应该由已经存在的字段组建，而不是人为产生的。虽然并不建议永远不要使用代理主键，但是建议你谨慎使用。如果发现实际上在每个表上都使用了 AUTO_INCREMENT，那么你可能过度使用了这个功能。

外键也是由于规范化程序而创建的，它们使表之间形成了一种父子或主次的关系，其中一个表中的一行是主，另一个表中的一行包含了这个主行的细节信息。外键是详细表中的一个字段，并指向主表中某个字段。外键还允许级联操作，即当删除主表的行时，附表中的相关行也被删除。



目前，只有 InnoDB 支持外键。

我们已经讨论了使用枚举替换小型查找表，但是，枚举值是个性能工具，这是因为包含枚举值的文本只在表头结构中保存一次。

MySQL 中集合的使用方法与枚举值的使用方法类似。然而，一组字段类型允许在集合中存储多个值，可以使用集合存储表示数据属性的信息，而不是使用主 / 从关系。这样不仅可以节省表空间（集合值按位组合），而且可以避免访问其他表中的值。（在行中节省下来的数字参考值形成了枚举值的索引（数组索引）。因此，枚举值列表可以节省空间，并可以使遍历数据更高效。一个枚举类型只允许存储一个值。）

339

使用 DEFAULT 选项为字段提供默认值是个很好的办法，可以防止出现错误结构的数据值。例如，如果表中有个数据型的字段，这个字段用于计算，你可能希望当这个字段未知时，用默认值代替之。可以在大部分数据类型上设置默认值，还可以在日期和时间字段上使用默认时间，这样可以避免无效日期时间值的产生。更重要的是，默认值可以允许应用程序不提供值（或者使用不太可靠的方法使用户提供值），从而在插入数据时减少发送到服务器的数据量。

在指定字段必须有值时，还应该使用 NOT NULL 选项。在执行 INSERT 时，如果没有给 NOT NULL 字段提供任何值，INSERT 语句将执行失败。这保证了数据的完整性，并确保所有重要字段都有值。

使用 EXPLAIN、ANALYZE和OPTIMIZE

前面已经讨论了这些命令的好处，在这里将这些命令罗列出来，以提醒你：这些工具在诊断和调优时很重要，在不发生错误的前提下经常使用它们，但是请小心使用。具体来说，当 ANALYZE 和 OPTIMIZE 有意义且不是作为定期的预定的事件时使用它们。我们发现有些系统管理员晚上使用这些命令，但是一般情况下，这样做是不值得的，并且会产生不必要的表副本（就像前面的实例）。显然，强制系统定期复制数据浪费时间，并会导致操纵过程中的访问有限。

现在已经讨论了如何监控和提高数据库性能，让我们看看最成功和最流行的功能——复制。下一节将讨论如何监控和提高 MySQL 中的复制。我们已经在上次讨论过这个主题，在你开始提高复制性能之前，你的服务器、数据库和查询的性能必须很好。

提高性能的最佳实践

诊断和改善数据库性能的详情在专门讨论这个主题的书上有介绍，事实上，这些内容覆盖了很多页。

为了本书内容的完整性，并作为一般的参考，本节包含了处理性能异常的一组最佳实践

方法，以便于参考。我们通过常见的问题给这些方法分组。

340

一切都慢

当整个系统运行很慢时，你应该专注于系统是如何运行的，从操作系统开始。可以使用下面介绍的一个或多个方法识别和改进系统的性能：

- 检查硬件问题；
- 改善硬件环境（例如添加硬盘）；
- 考虑将数据迁移到独立的磁盘上；
- 检查操作系统的配置是否正确；
- 考虑将有些应用迁移到其他服务器上；
- 考虑可以向外扩展的复制；
- 优化服务器性能。

慢查询

可以使用下面的方法改善慢查询日志中的任何查询，或改善那些被认为有问题的查询：

- 规范化数据库模式；
- 使用 EXPLAIN 识别丢失或不正确的索引；
- 使用 benchmark() 函数测试部分查询；
- 考虑重写查询；
- 对标准查询使用视图；
- 启用 Query Cache。



复制 Slave 不会将复制查询写入慢查询日志，不论这个查询是否会被写入 Master 中的慢查询日志中。

慢应用

如果应用程序出现性能问题，你应该检查应用组件，以确认问题出在哪里。也许你会发现只有一个模块导致问题的发生，但是有时候也可能更严重。下面介绍的方法可以帮助你识别和解决应用程序问题：

- 开启 Query Cache；
- 考虑并优化存储引擎；

- 确认是否是服务器或操作系统的问题；
- 定义应用程序的基准，并将它与已知基准比较；
- 检查内部（在应用程序中编写的）查询，并最大化它们的性能；
- 分而治之——一次只检查一个部分；
- 使用划分来分散数据；
- 检查各个分区的索引。

341

慢复制

如前所述，与复制相关的性能问题，通常与数据库和服务器性能问题隔离开，在诊断复制的性能问题时，使用下面介绍的方法：

- 确保你的网络运行状况最佳；
- 确保服务器配置正确；
- 优化数据库；
- 限制 Master 的更新；
- 将数据读取划分到多个 Slave 中；
- 检查 Slave 的复制延迟；
- 定期维护日志（二进制日志和中继日志）；
- 在带宽有限的情况下，使用压缩；
- 使用包容性和排他性日志选项，最小化复制的内容。

小结

监控 MySQL 服务器有很多事情需要做，我们已经讨论了用于监控服务器的基本 SQL 命令、`mysqladmin` 命令行实用程序、benchmark 套件、MySQL 管理器和 MySQL 查询浏览 GUI 工具。还介绍了提高数据库性能的一些最佳实践。

现在你已经了解了基本的操作系统监控、数据库性能、MySQL 监控和 benchmarking，且已经拥有能够成功优化服务器性能的工具和知识。

342

Joel 一边笑，一边写有关 Susan 的嵌套查询问题的报告，这需要花费数个小时才能找出问题，但是在他向开发者解释查询的开销后，他们同意更改查询，并使用存储在内存表中的查找表。Joel 觉得老板将会很满意他的构思，他正点击发送时，老板正巧出现在他的办公室门口。

“Joel！”

Joel 吓了一跳，尽管知道 Summerson 先生在那儿。

“我已经解决了营销部门的应用程序的问题，先生。”他快速地说道。

“太好了，我很想知道你是如何解决这个问题的。”

Joel 不确定老板是否了解他邮件中提到的技术，但是他知道，如果不向老板解释，老板也会问的。

Summerson 先生点了点头，然后走了。Joel 打开了来自于西雅图的 Phil 的邮件，这个邮件是关于抱怨复制问题的。Joel 马上意识到问题远比他现在了解的要严重。

监控存储引擎

Joel 正在喝拿铁，电话铃响了。他吓了一跳，因为从早上到现在电话还没响过。他拿起电话筒，听到了发动机的噪声。他以为电话打错了，迟疑地说："喂？"

"Joel！联系上你太好了！"原来是 Sunnerson 先生从车里打过来的。

"是的，先生。"

"我正在去机场的路上，去接见西雅图办事处的销售人员。我想请你看看新应用的数据库。西雅图的开发人员认为我们需要找到一种更好的配置以提高性能。"

Joel 曾想过会有这种事情发生。他了解一点 MyISAM 和 InnoDB，但是并不熟悉监控，更不了解性能调优。"我会查看的，先生。"

"好极了，谢谢你，Joel，我会发电子邮件给你。"Joel 还没来得及回话电话就被挂断了。Joel 喝完了最后一点拿铁，然后开始阅读 MySQL 有关存储引擎的知识。

现在你已经知道服务器何时运行良好（以及何时运行不好），那么怎么知道存储引擎的运行情况如何呢？如果你正在执行一个或多个数据库事务，或者你为了快速查询而需要存储引擎高速运行，将需要监控存储引擎。本章讨论高级监控，重点关注监控和提高存储引擎的性能，主要介绍两个最流行的存储引擎：MyISAM 和 InnoDB。

MySQL 一个非常强大和独特的功能是支持多种存储引擎。虽然 MySQL 没有内置通用存储引擎监控，甚至也没有标准的监控功能，但是你可以监控和配置（调优）最流行的存储引擎的性能。

344 本章将介绍 MyISAM 和 InnoDB 存储引擎，重点讨论如何监控它们，并为如何改善性能提供一些实用的建议。

MyISAM

MyISAM 存储引擎需要监控的信息很少。这是因为 MyISAM 存储引擎是建立在 Web 应用程序上的，主要致力于快速查询，因此，对于该存储引擎，你只需要调节服务器上的一个功能——Key cache。这并不意味着没有其他措施可以用来提高性能，相反，有很多事情可以做。提高性能的方法大致分为三类：优化磁盘存储、通过监控和优化 Key cache 来有效地使用内存，以及优化数据库表。

主要从以下几个方面来讨论如何提高性能：

- 优化磁盘存储；
- 优化数据库表；
- 使用 MyISAM 实用工具；
- 按照索引顺序存储表；
- 压缩表；
- 对数据表进行碎片整理；
- 监控 Key Cache；
- 预加载 Key Cache；
- 使用多个 key Cache；
- 其他参数的考虑。

下面将逐个简单介绍。

优化磁盘存储

MyISAM 的磁盘空间优化是系统配置项，而非 MyISAM 特有的调优参数。MyISAM 中的数据表有自己的存储形式，即 *.myd*（数据文件）和一个或多个 *.myi*（索引）文件。这些文件与 *.frm* 文件一起存储在以数据库名命名的文件目录下，该目录由 `--datadir` 启动选项指定。因此，MyISAM 的磁盘空间优化方法与服务器上的磁盘空间优化方法相同。也就是说，可以将数据目录移到其所在磁盘上以提高数据库性能，还可以使用 RAID 或其他高可用性存储磁盘来进一步提高性能。

345 ## 优化数据库表

可以使用以下几种 SQL 命令优化数据库表：`ANALYZE TABLE`、`OPTIMIZE TABLE` 和

REPAIR TABLE。

ANALYZE TABLE 命令用于检测和重组表的关键字分布情况。当通过字段而非常量的方式进行表连接时，MySQL 通过关键字的分布情况决定表的连接顺序。关键字分布还决定了查询时所使用的索引。前面的“使用 ANALYZE TABLE”一节中已经详细地介绍了该命令，这里就不再赘述了。

REPAIR TABLE 命令其实不是一个优化工具，可以使用它为 MyISAM、Archive 和 CVS 存储引擎恢复崩溃的表。该命令用于恢复那些崩溃的或运行很慢的表（这通常表明该表已经退化，需要重组或修复）。



REPAIR TABLE 命令与 myisamchk --recover<table name> 命令功能相同。

OPTIMIZE TABLE 命令用于恢复被删除的块和重组表，从而提高数据库性能。可以在 MyISAM、BDB 和 InnoDB 表上使用该命令。

尽管这些命令很实用，但是还有很多高级工具可以用于管理 MyISAM 表。

使用MyISAM实用工具

MySQL 发布包中包含了很多管理 MyISAM 存储引擎（表）的专用工具。

- myisam_ftdump：显示全文索引信息
- myisamchk：执行 MyISAM 表的分析
- myisamlog：查看 MyISAM 表的更改日志
- myisampack：压缩表以减少存储量

myisamack 是监控 MyISAM 的主力工具，用于显示 MyISAM 表的信息，并对 MyISAM 表进行分析、修复和优化。可以在一个或多个表上运行该命令，但是只能在脱机状态（即关闭表和服务器）下使用。



在运行该工具前，请务必备份表，防止修复或优化失败。这样可以避免表损坏或不可修复。

346

有关性能改进的选项说明如下。参看 MySQL 的网上参考手册，以了解这些可选项的完整说明。

- analyze
 分析索引的关键字分布以提高查询性能。
- backup
 在改变表之前复制一份副本（即 *.myd* 文件）。
- check
 检查表的错误信息并报告之。
- extended-check
 彻底检查表（包括索引）的错误信息，并报告之。
- force
 查找错误并修复之。
- information
 显示表的统计信息。在运行 recover 恢复表之前，使用该命令查看表的状况。
- medium-check
 更深入地检查表，并修复之。这比 extended-check 检查的信息少。
- recover
 全面修复表（修复数据结构）。执行除了唯一键重复的所有修复工作。
- safe-recover
 执行传统形式的修复，即有序地读取所有行，并更新所有索引。
- sort-index
 降序排列索引树。这样能够减少索引结构的查找时间，加快索引的访问速度。
- sort records
 按指定的索引序列排序记录信息。这样可以提高某些基于索引的查询的性能。

图例 9-1 展示了运行 myisamchk 命令后显示的 MyISAM 表信息。

例9-1：myisamchk工具

```
MyISAM file:            /usr/local/mysql/data/employees/employees
Record format:          Packed
347 Character set:      latin1_swedish_ci (8)
File-version:           1
Creation time:          2009-12-03 20:52:12
Status:                 changed
Data records:                 297024 Deleted blocks: 3000
Datafile parts:               300024 Deleted data: 95712
Datafile pointer (bytes):          6 Keyfile pointer (bytes): 6
Datafile length:             9561268 Keyfile length: 3086336
Max datafile length:     281474976710654
Max keyfile length:     288230376151710719
Recordlength:                     44
```

```
table description:
Key   Start   Len   Index    Type   Rec/key        Root     Blocksize
1     1       4     unique   long         1     2931712         1024
```

按索引顺序存储表

按照索引顺序存储表数据可以提高大量数据范围查询（例如 WHERE a > 5 AND a < 15）的效率。这种排序允许查询时有序地访问数据，而无须搜索数据的磁盘页。如果需要按照索引顺序排序表，可以使用 myisamchk 工具排序记录选项（-R），并指定所使用的索引，索引编号从 1 开始。下面的命令按照第二个索引的顺序对 test 数据库的表 table1 进行排序。

```
myisamchk -R 2 /usr/local/mysql/data/test/table1
```

也可以使用 ALTER TABLE 和 ORDER BY 命令实现同样的效果。

当在数据表中添加新行时，这样排序表的方式无法保证表数据仍按索引顺序存储。删除操作不影响排序，但是当添加新行时，表将变得无序，导致数据库性能下降。如果在经常变更的表上采用按索引顺序存储表的技术来提高性能，可能需要定期运行该命令以确保表的存储顺序最佳。

压缩表

压缩数据库可以节约空间。虽然 MySQL 中压缩数据的方法很多，但是 MyISAM 存储引擎只能压缩只读表，因为 MyISAM 不能解压、重新排序，也不能对新增（或删除）数据进行压缩。MyISAM 中使用 myisampack 来压缩表，如下所示：

```
myisampack -b /usr/local/mysql/data/test/table1
```

在压缩数据表前，一般使用备份选项（-b）来创建表的备份副本。这样可以在不需要重新运行 myisampack 命令的情况下，使表变为可写的。

为什么要压缩只读表？有两个原因。首先，它可以为易于压缩的数据（如文本）节约存储空间。其次，当查询读取压缩后的表，并通过主键或唯一索引来查找表中数据行时，在比较其他条件之前，仅对单行数据进行解压缩。 348

myisampack 命令有许多选项。如果你对压缩只读表有兴趣，请参阅 MySQL 在线文档，以了解怎样操控该命令的压缩功能。

对数据表进行碎片整理

当 MyISAM 数据表有很多删除和插入操作时，其物理存储将变得很零散。例如，已删除

数据带来物理存储上的空缺，而插入操作可能破坏原来的存储顺序。为了优化数据表，将其重新组织为期望的顺序和形式，可以使用 OPTIMIZE TABLE 命令或 myisamchk 实用工具。

对于有指定排序的表而言，应该定期运行这些命令，以确保这些表的存储形式最优。另外，如果某段时间内数据进行了很多更新，也应该运行这些命令。

监控Key Cache

MySQL 的 Key Cache 是一个高效的存储结构，用于存储频繁使用的索引数据。Key Cache 仅用于 MyISAM，并使用快速查找机制（通常是 B- 树）存储关键字。索引作为链接列表存储在内存中，可以被快速检索到。Key Cache 在第一个 MyISAM 数据表被读取时自动创建。每次查询 MyISAM 数据表前，都会检查一遍 Key Cache。如果在缓存中找到索引，则直接在内存中执行索引检索，而不需要先从磁盘上读取索引。Key Cache 是使 MyISAM 的快速查询比其他存储引擎都快的秘密武器。

MyISAM 中有许多变量用于控制 Key Cache，可以使用 SHOW VARIABLES 和 SHOW STATUS 命令或 MySQL 管理器监控这些变量。我们在第 8 章（见图 8-5）已经讨论了 MySQL 管理器的内置 Key Cache 监控器。例 9-2 显示了用 SHOW 命令监控这些变量的实例。

例9-2：主缓存的状态和系统变量

```
mysql> SHOW STATUS LIKE 'Key%';
+------------------------ +-------+
| Variable_name           | Value |
+------------------------ +-------+
| Key_blocks_not_flushed  | 0     |
| Key_blocks_unused       | 6694  |
| Key_blocks_used         | 0     |
| Key_read_requests       | 0     |
| Key_reads               | 0     |
| Key_write_requests      | 0     |
| Key_writes              | 0     |
+-------------------------+-------+
7 rows in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'key%';
+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| key_buffer_size          | 8384512 |
| key_cache_age_threshold  | 300     |
| key_cache_block_size     | 1024    |
| key_cache_division_limit | 100     |
+--------------------------+---------+
```

349

```
4 rows in set (0.01 sec)
```

如你所想，Key Cache 非常复杂。所以 Key Cache 调优将是一个挑战。建议监控 Key Cache 的利用率并根据情况改变 Key Cache 的大小，最好不要改变其运行方式，因为默认配置已能够满足运行要求。

如果想提高缓存命中率，可以使用以下两种方法：(1) 预加载缓存；(2) 使用多个 Key Cache 并为默认 Key Cache 分配更多的内存。

预加载Key Cache

将索引预加载到 Key Cache 可以加快查询速度，因为索引已经加载到缓存，并且是按顺序加载的（而不是随机的，例如并发操作下的 Key Cache 就是随机加载的）。然而，必须保证缓存中有足够的空间存储索引。对某些特定应用或使用模型来说，预加载是提高查询速度的有效方法。比如，如果在一个应用程序（例如典型的工资审计程序）执行过程中，某个特定的表被查询很多次，这时你可以将该表的相关索引预加载到 Key Cache 中，从而提高数据库性能。使用 LOAD INDEX 命令执行预加载功能，如例 9-3 所示：

例 9-3：预加载索引到主内存中

```
mysql> LOAD INDEX INTO CACHE salaries IGNORE LEAVES;
+--------------------+--------------+----------+----------+
| Table              | Op           | Msg_type | Msg_text |
+--------------------+--------------+----------+----------+
| employees.salaries | preload_keys | status   | OK       |
+--------------------+--------------+----------+----------+
1 row in set (1.49 sec)
```

该例将 salary 表的索引加载到 Key Cache 中，IGNORE LEAVES 语句表明只预加载索引的非叶子节点。虽然没有特殊的命令用于刷新 Key Cache，但是可以通过修改表（例如重组索引或删除并重建索引）强行从 Key Cache 中移除相关索引。 350

使用多个Key Cache

MySQL 有一个鲜为人知的高级特性，即创建多个 Key Cache 或自定义 Key Cache，以减少对默认 Key Cache 的争用。该特性允许将一个或多个表的索引加载到自定义的特殊缓存中。这意味着按任务分配内存，需要认真规划。如果某段时间内对一组表执行大量查询操作，且频繁引用这些表上的索引，那么使用这种策略将大大提高数据库系统的性能。

要创建一个二级 Key Cache，首先需要使用 SET 命令分配内存，然后执行一个或多个 CACHE INDEX 命令去加载表的索引。与默认 Key Cache 不同的是，可以通过将二级 Key Cache 的大小设为 0 将其刷新或移除。例 9-4 显示了如何创建二级 Key Cache，然后添加

表索引到缓存中。

例9-4：使用二级主缓存

```
mysql> SET GLOBAL emp_cache.key_buffer_size=128*1024;
Query OK, 0 rows affected (0.00 sec)

mysql> CACHE INDEX salaries IN emp_cache;
+-------------------+--------------------+----------+----------+
| Table             | Op                 | Msg_type | Msg_text |
+-------------------+--------------------+----------+----------+
| employees.salaries | assign_to_keycache | status   | OK       |
+-------------------+--------------------+----------+----------+
1 row in set (0.00 sec)

mysql> SET GLOBAL emp_cache.key_buffer_size=0;
Query OK, 0 rows affected (0.00 sec)
```

请注意，二级 Key Cache 定义一个名为 `emp_cache` 的新变量，并将其大小设置为 128KB。这是 `SET` 命令的一个特定语法，虽然看起来像是创建一个新的系统变量，但实际上，是创建了一个新的全局用户变量。可以通过以下方式确定一个二级缓存是否存在或其大小：

```
mysql> select @@global.emp_cache.key_buffer_size;
+-----------------------------------+
| @@global.emp_cache.key_buffer_size |
+-----------------------------------+
|                            131072 |
+-----------------------------------+
1 row in set (0.00 sec)
```

351

由于二级 Key Cache 是全局的，因此只有将其大小设置为 0 或重新启动服务器时，才存在二级 Key Cache。



可以通过以下方法保存多个 Key Cache 的配置情况：将配置语句保存在一个文件中，然后在系统启动时，使用 MySQL 选项文件的 *[mysql]* 部分的 `init-file=<path_to_file>` 命令执行该文件。

其他需要考虑的参数

还有其他一些参数需要考虑。请记住，一次只能改变一个参数，并且只有当理由充分时才需要改变参数。在没有好的理由和合理的期望结果的情况下，永远不要改变复杂项的配置，如存储引擎的配置。

- myisam_data_pointer_size
 未指定 MAX_ROW（表中存储的最大行数）值的情况下，创建表时使用的默认指针大小一般取 2~7（单位为字节），默认值为 6。
- myisam_max_sort_file_size
 数据排序时使用的临时文件大小的最大值。增大该值可以加速索引的修复和重组。
- myisam_recover_options
 MyISAM 的数据恢复模式，可用于 OPTIMIZE TABLE。该模式包括默认、备份、强制、快速，这些选项可以任意组合。Default：执行没有备份、强制或快速查询的恢复。备份：在恢复前首先创建备份。Force：即使数据丢失多行，仍进行数据恢复。Quick：如果没有块标记为删除，此时将不检测表中的行。请根据恢复的严重性决定使用哪些项。
- myisam_repair_threads
 如果该值大于 1，则并行执行修复和排序操作，从而加快操作速度。否则，这些操作将顺序执行。
- myisam_sort_buffer_size
 排序操作的缓存区大小。增加该值有助于排序索引。然而，如果该值超过 4GB，则只适用于 64 位机器。
- myisam_stats_method
 在统计操作中用于控制服务器如何计算索引值分布中的 NULL 值。这也会影响到优化，请小心使用。
- myisam_use_mmap
 为读写 MyISAM 表开启存储器映射（memory map）选项。当许多小写入与大数据集查询并行进行时，该功能将非常有用。

352

我们已经讨论了一些关于监控和改进 MyISAM 性能的策略。虽然这些讨论比较简单，但是其覆盖了高效使用 MyISAM 的最重要部分。请参看 MySQl 在线参考手册，以获得关于键缓存和 MyISAM 存储引擎的更多信息。

MySQL 复制和高可用性

MyISAM 数据损坏的概率比 InnoDB 高，因此，MyISAM 需要较长的恢复时间。此外，由于 MyISAM 不支持事务，一次只能执行一个事件，这可能导致事务中只有部分语句被执行。另外，由于 Slave 是单线程执行的，在执行长时间运行的查询时，Slave 可能会滞后，因此，在一个高可用性解决方案中，在 Slave 上使用 MyISAM 将会出问题。

InnoDB

InnoDB 存储引擎有很多优化选项，每一个选项的深入讲解都足以覆盖一本书。例如，用于控制 InnoDB 行为的变量有 50 个，还有 40 多个状态变量用于表示性能和状态相关的元数据。本节将讨论如何监控 InnoDB 存储引擎，重点讨论提高性能的关键问题。

对于提高性能的方法，这里主要讨论以下几个方面：

- 使用 SHOW ENGINE 命令；
- 使用 InnoDB 监控器；
- 监控日志文件；
- 监控缓冲池；
- 监控表空间；
- 使用 INFORMATION_SCHEMA 表；
- 考虑其他参数。

下面将简单讨论这些问题。但是，在开始讨论前，让我们先简单了解一下 InnoDB 的架构特性。

353 InnoDB 存储引擎的架构非常复杂，是专门为高并发性和复杂事务性活动而设计的。它有许多高级功能，应该在改进性能前优先考虑这些功能。我们主要关注那些可以被监控和改进的功能，包括索引、缓冲池、日志文件和表空间。

InnoDB 表使用聚集索引。即使未指定索引，InnoDB 也会为每行分配一个内部值，用于使用聚集索引。聚集索引是一种数据结构，它不仅存储索引，还存储数据本身。也就是说，一旦定位到索引中的某个值，就可以直接检索数据而无须额外的磁盘寻道。当然，主键索引或者表的第一个唯一索引都采用聚集索引创建。

如果创建了二级索引，聚集索引的关键字（主关键字、唯一键或者行 ID）信息都会存在二级索引中。这样可以快速重新定位和回访聚集索引中的原始数据。这也意味着如果使用主关键字列扫描二级索引，则只需要用二级索引来检索数据。

缓冲池是用于管理事务和读写磁盘数据的缓存机制，如果配置得当，将会提高磁盘访问速率。缓冲池同时还是崩溃恢复的一个重要组成部分，因为缓冲池内的信息将会定期被写入磁盘（例如关机时）。缓冲池是一个内存组件，必须监控其有效性，保证配置的正确性。

InnoDB 也使用缓冲池来存储数据变更和事务。InnoDB 通过将数据变更保存到缓冲池中的数据页（块）中进行缓存。每次引用数据页都会放到缓冲池中，发生改变后就标记为“dirty”。然后，这个改变被写入到磁盘中以更新数据，并向日志中写入一个副本。这些

日志文件的名字为 *ib_logfile* 或 *ib_logfile*1。可以在 MySQL 服务器的数据目录中看到这些文件。

InnoDB 存储引擎使用两种基于磁盘的机制存储数据：日志文件表空间。在关机或死机之前，InnoDB 还使用这些日志来重建（或重做）数据变更。在程序启动时，InnoDB 读取日志并自动将脏数据写入磁盘，从而在系统崩溃前可以恢复缓冲中的数据变更。

表空间是一个组织工具，在 InnoDB 中作为独立于机器的文件使用，包括数据、索引及回滚机制（回滚事务）。默认情况下，所有表共享一个表空间。也可以将表存储在它们自己的表空间中。这些表空间同时包含数据和表的索引。表空间可以自动扩展成多个文件，从而使你可以在表中存储更多的数据，该数据量多于操作系统可以处理的数据量。你还可以将表空间划分为多个文件，然后存储在不同的磁盘上。



使用 innodb_file_per_table 为每个表创建单独的表空间。在设置该选项之前创建的表将存储在共享表空间中。使用此命令只影响新创建的表。 354

使用SHOW ENGINE命令

SHOW ENGINE INNODB STATUS 命令显示有关 InnoDB 存储引擎状态的统计和配置信息。这是查看 InnoDB 信息的标准方法。该命令显示的统计列表很长且信息全面。例 9-5 节选了在标准安装的 MySQL 上该命令的运行情况。

例 9-5：SHOW ENGINE INNODB STATUS 命令

```
mysql> SHOW ENGINE INNODB STATUS \G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
091205 18:31:10 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 1 seconds
----------
BACKGROUND THREAD
----------
srv_master_thread loops: 233 1_second, 34 sleeps, 23 10_second,
                         3 background, 3 flush
srv_master_thread log flush and writes: 44 log writes only: 448
----------
SEMAPHORES
----------
```

```
OS WAIT ARRAY INFO: reservation count 882, signal count 901
Mutex spin waits 2501, rounds 21869, OS waits 388
RW-shared spins 165, OS waits 144; RW-excl spins 0, OS waits 335
Spin rounds per wait: 8.74 mutex, 26.70 RW-shared, 10301.00 RW-excl
------------
TRANSACTIONS
------------
Trx id counter 2969
Purge done for trx's n:o < 2519 undo n:o < 0
History list length 3
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started, OS thread id 4491317248
MySQL thread id 4, query id 152 localhost root
SHOW ENGINE INNODB STATUS
---TRANSACTION 2968, ACTIVE 0 sec, OS thread id 4548612096 inserting
mysql tables in use 1, locked 1
7 lock struct(s), heap size 1216, 1171 row lock(s), undo log entries 11375
MySQL thread id 3, query id 151 localhost root update
INSERT INTO 'salaries' VALUES (204383,71223,'1998-11-14','1999-09-07'),
...
--------
FILE I/O
--------
I/O thread 0 state: waiting for i/o request (insert buffer thread)
I/O thread 1 state: waiting for i/o request (log thread)
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (read thread)
I/O thread 4 state: waiting for i/o request (read thread)
I/O thread 5 state: waiting for i/o request (read thread)
I/O thread 6 state: waiting for i/o request (write thread)
I/O thread 7 state: waiting for i/o request (write thread)
I/O thread 8 state: waiting for i/o request (write thread)
I/O thread 9 state: waiting for i/o request (write thread)
Pending normal aio reads: 0, aio writes: 0,
ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 1
31 OS file reads, 3979 OS file writes, 1593 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 146.85 writes/s, 78.92 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2,
0 inserts, 0 merged recs, 0 merges
Hash table size 276707, node heap has 444 buffer(s)
92903.10 hash searches/s, 459.54 non-hash searches/s
---
LOG
---
Log sequence number 219912928
Log flushed up to 218951284
```

355

```
Last checkpoint at 217528539
0 pending log writes, 0 pending chkp writes
1074 log i/o's done, 46.95 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 138805248; in additional pool allocated 0
Dictionary memory allocated 70560
Buffer pool size 8192
Free buffers 760
Database pages 6988
Modified db pages 113
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 21, created 6968, written 10043
0.00 reads/s, 89.91 creates/s, 125.87 writes/s
Buffer pool hit rate 1000 / 1000
LRU len: 6988, unzip_LRU len: 0
I/O sum[9786]:cur[259], unzip sum[0]:cur[0]
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
1 read views open inside InnoDB
Main thread id 4528414720, state: sleeping
Number of rows inserted 2078594, updated 0, deleted 0, read 0
31059.94 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

356

SHOW ENGINE INNODB MUTEX 命令展示了 InnoDB 的互斥体信息，对存储引擎中的线程调优很有帮助。例 9-6 节选了在标准安装的 MySQL 上该命令的运行情况。

例 9-6：SHOW ENGINE INNODB MUTEX 命令

```
mysql> SHOW ENGINE INNODB MUTEX;
+--------+--------------------+---------------+
| Type   | Name               | Status        |
+--------+--------------------+---------------+
| InnoDB | trx/trx0rseg.c:167 | os_waits=1    |
| InnoDB | trx/trx0sys.c:181  | os_waits=7    |
| InnoDB | log/log0log.c:777  | os_waits=1003 |
| InnoDB | buf/buf0buf.c:936  | os_waits=8    |
| InnoDB | fil/fil0fil.c:1487 | os_waits=2    |
| InnoDB | srv/srv0srv.c:953  | os_waits=101  |
| InnoDB | log/log0log.c:833  | os_waits=323  |
+--------+--------------------+---------------+
7 rows in set (0.00 sec)
```

上例中的 Name 列显示了创建互斥体的源文件和行号。Status 列显示了互斥体在操作系统上等待的次数（例如 os_waits=5）。如果源代码是使用 UNIV_DEBUG 指令编译的，该列的值可能是：

- count
 请求互斥体的次数。
- spin_waits
 自旋锁运行的次数。
- os_waits
 互斥体在操作系统上等待的次数。
- os_yields
 线程放弃时间片并返回操作系统的次数。
- os_wait_times
 互斥体等待操作系统的总时间。

SHOW ENGINE INNODB STATUS 命令显示了直接来自 InnoDB 存储引擎的很多信息。虽然这些信息没有格式化（即不是整齐的行和列），但是有许多工具可以将它们重新排列显示。例如，InnoTop（见前面的“InnoTop”）命令就是使用该方法传达数据的。

357

使用InnoDB监控器

InnoDB 存储引擎是唯一支持直接监控的本地存储引擎。InnoDB 背后有一个称为监控器（*monitor*）的特殊机制，它为服务器和客户端工具收集和报告统计信息。下面各项（以及大多数第三方工具）都与 InnoDB 监控工具交互，因此 InnoDB 可以通过 MySQL 服务器监控下列项：

- 表和记录锁；
- 锁等待；
- 信号量等待；
- 文件 I/O 请求；
- 缓冲池；
- 清除和插入缓冲合并活动。

通过 SHOW ENGINE INNODB STATUS 命令自动连接 InnoDB 监控器，并显示监控器产生的信息。不过，还可以在 MySQL 中通过创建特殊的表，直接从 InnoDB 监控器得到这些信息。这些表的真实结构及存储在什么地方并不重要（如果使用 ENGINE=INNODB 语句），一旦创建，每个表中的数据都会转储到 stderr 中。可以通过错误日志或 MySQL 控制台（使用 --console 选项启动 MySQL）看到这些信息。要启动 InnoDB 监控器，请在数据

库中创建以下表：

```
mysql> SHOW TABLES LIKE 'innodb%';
+---------------------------+
| Tables_in_test (innodb%)  |
+---------------------------+
| innodb_lock_monitor       |
| innodb_monitor            |
| innodb_table_monitor      |
| innodb_tablespace_monitor |
+---------------------------+
4 rows in set (0.00 sec)
```

要关闭监控器，只需删除该表即可。监控器每隔 15 秒自动重新生成数据。

每个监控器提供以下数据：

- `innodb_monitor`
 标准监控器与 SQL 命令显示的信息相同。如例 9-5 所示的监控器输出结果的实例。SQL 命令与 `innodb_monitor` 之间唯一的不同是：`innodb_monitor` 将结果以格式化的形式输出到 `stderr`，且与 MySQL 客户端垂直展示的信息格式相同。
- `innodb_lock_monitor`
 锁监控器也与 SQL 命令显示的信息相同，但是还包括关于锁的其他信息。
- `innodb_table_monitor`
 表监控器生成内部数据字典的详细报告。例 9-7 显示了该报告的一个摘录（为便于阅读，这里进行了格式化）。请注意每个表的扩展数据，其中包括字段定义、索引、行号、外键及其他更多的信息。在诊断表问题或者想了解索引的细节信息时，使用该报告。

358

例 9-7：InnoDB 表监控器报告

```
=========================================
091208 21:10:06 INNODB TABLE MONITOR OUTPUT
=========================================
-------------------------------------
TABLE: name sakila/address, id 0 14, flags 1,
       columns 11, indexes 2, appr.rows 628
COLUMNS: address_id: DATA_INT DATA_UNSIGNED DATA_BINARY_TYPE
                     DATA_NOT_NULL len 2;
         address: type 12 DATA_NOT_NULL len 150;
         address2: type 12 len 150;
         district: type 12 DATA_NOT_NULL len 60;
         city_id: DATA_INT DATA_UNSIGNED DATA_BINARY_TYPE
                  DATA_NOT_NULL len 2;
...
  INDEX: name PRIMARY, id 0 17, fields 1/10, uniq 1, type 3
```

```
   root page 52, appr.key vals 628, leaf pages 4, size pages 5
   FIELDS: address_id DB_TRX_ID DB_ROLL_PTR address address2
           district city_id postal_code phone last_update
  INDEX: name idx_fk_city_id, id 0 18, fields 1/2, uniq 2,
         type 0 root page 53, appr.key vals 599, leaf pages 1,
         size pages 1
   FIELDS: city_id address_id
  FOREIGN KEY CONSTRAINT sakila/fk_address_city:
                         sakila/address ( city_id )
              REFERENCES sakila/city ( city_id )
...
-----------------------------------
END OF INNODB TABLE MONITOR OUTPUT
===================================
```

- innodb_tablespace_monitor

 显示共享表空间的扩展信息，包括文件段的列表。它还验证表空间分配的数据结构。该报告可能相当详细且很长，因为它列出了表空间的所有细节信息。例 9-8 显示了该报告的摘录。

359 例 9-8：InnoDB 表空间监控器报告

```
=============================================
091208 21:14:19 INNODB TABLESPACE MONITOR OUTPUT
=============================================
FILE SPACE INFO: id 0
size 16000, free limit 15424, free extents 2
not full frag extents 3: used pages 144, full frag extents 41
first seg id not used 0 714
SEGMENT id 0 1 space 0; page 2; res 2 used 2; full ext 0
fragm pages 2; free extents 0; not full extents 0: pages 0

...

SEGMENT id 0 411 space 0; page 209; res 29 used 29; full ext 0
fragm pages 29; free extents 0; not full extents 0: pages 0
SEGMENT id 0 412 space 0; page 209; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 0 413 space 0; page 209; res 96 used 60; full ext 0
fragm pages 32; free extents 0; not full extents 1: pages 28
SEGMENT id 0 414 space 0; page 209; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 0 415 space 0; page 209; res 96 used 33; full ext 0
fragm pages 32; free extents 0; not full extents 1: pages 1
NUMBER of file segments: 275
Validating tablespace
Validation ok
---------------------------------------
END OF INNODB TABLESPACE MONITOR OUTPUT
=======================================
```

可见，InnoDB 监控器报告很多详细信息。请长时间保持其处于开启状态，这样可以向日志文件添加大量信息。

监控日志文件

InnoDB 日志文件在你的数据和操作系统之间缓冲数据，这些文件正常运行可以确保系统性能良好。还可以通过查看以下系统状态变量，直接监控这些日志文件。

```
mysql> SHOW STATUS LIKE 'Innodb%log%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_log_waits              | 0     |
| Innodb_log_write_requests     | 0     |
| Innodb_log_writes             | 2     |
| Innodb_os_log_fsyncs          | 5     |
| Innodb_os_log_pending_fsyncs  | 0     |
| Innodb_os_log_pending_writes  | 0     |
| Innodb_os_log_written         | 1024  |
+-------------------------------+-------+
```

我们已经看到了由 InnoDB 监控器显示的信息，但是也可以使用以下状态变量获取日志文件的详细信息。 360

- `Innodb_log_waits`
 当日志文件太小（即没有足够空间存储所有数据）时，操作必须等待日志刷新的等待时间计数器。如果该值开始增加并长期大于零（除批量操作以外），可以增加日志文件的大小。
- `Innodb_log_write_requests`
 日志写入请求的数量。
- `Innodb_log_writes`
 数据被写入日志的次数。
- `Innodb_os_log_fsyncs`
 操作系统文件同步的数量（即 `fsync()` 方法调用）。
- `Innodb_os_log_pending_fsyncs`
 阻塞（pend）的文件同步请求的数量。如果该值开始增加并长期大于零，可能需要检查磁盘访问问题。
- `Innodb_os_log_pending_writes`
 阻塞（pend）的日志写请求的次数。如果该值开始增加并长期大于零，可能需要检查磁盘访问问题。
- `Innodb_os_log_written`
 写到日志中的字节总量。

所有这些选项显示的都是数字信息，可以在 MySQL 管理器中建立描述这些信息的自定义图表。

监控缓冲池

缓冲池是 InnoDB 缓存频繁访问数据的地方，对缓冲池内数据的任何更新也会被缓存。缓冲池还存储当前事务的相关信息。因此，缓冲池是关乎性能的关键机制。

使用 SHOW ENGINE INNODB STATUS 命令查看缓冲池的相关信息，如例 9-5 所示。为了方便查看，我们在这里重新讲解缓冲池和内存的知识。

361

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 138805248; in additional pool allocated 0
Dictionary memory allocated 70560
Buffer pool size 8192
Free buffers 760
Database pages 6988
Modified db pages 113
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 21, created 6968, written 10043
0.00 reads/s, 89.91 creates/s, 125.87 writes/s
Buffer pool hit rate 1000 / 1000
LRU len: 6988, unzip_LRU len: 0
I/O sum[9786]:cur[259], unzip sum[0]:cur[0]
```

下面给出了该报告中需要重点注意的信息。我们将在后面更详细地讨论具体的状态变量。

- *Free buffers*
 空的且可用于缓冲数据的缓冲段个数。
- *Modified pages*
 发生变化（脏）的页数。
- *Pending reads*
 等待中的读请求个数，该值应该保持在低水平。
- *Pending writes*
 等待中的写请求个数，该值应该维持在低水平。
- *Hit rate*
 成功访问缓冲区的请求个数与总请求个数之间的比例，这个比值最好接近 1 ： 1。

还可以查看更多状态变量的信息。下面显示了 InnoDB 缓冲池的状态变量：

```
mysql> SHOW STATUS LIKE 'Innodb%buf%';
```

```
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| Innodb_buffer_pool_pages_data     | 21    |
| Innodb_buffer_pool_pages_dirty    | 0     |
| Innodb_buffer_pool_pages_flushed  | 1     |
| Innodb_buffer_pool_pages_free     | 8171  |
| Innodb_buffer_pool_pages_misc     | 0     |
| Innodb_buffer_pool_pages_total    | 8192  |
| Innodb_buffer_pool_read_ahead_rnd | 0     |
| Innodb_buffer_pool_read_ahead_seq | 0     |
| Innodb_buffer_pool_read_requests  | 558   |
| Innodb_buffer_pool_reads          | 22    |
| Innodb_buffer_pool_wait_free      | 0     |
| Innodb_buffer_pool_write_requests | 1     |
+-----------------------------------+-------+
```

缓冲池有很多状态变量用于显示关于缓冲池性能的主要统计信息，如缓冲池中的页状态、缓冲池的读写信息，以及缓冲池中读写等待的频率。下面详细地介绍了各个状态变量：

362

- Innodb_buffer_pool_pages_data
 含有数据的页数，包括不变和改变的页（即脏页）。
- Innodb_buffer_pool_pages_dirty
 改变的页（即脏页）的数目。
- Innodb_buffer_pool_pages_flushed
 缓冲池页面被刷新的次数。
- Innodb_buffer_pool_pages_free
 空页面的数目。
- Innodb_buffer_pool_pages_misc
 InnoDB 引擎用于管理工作的页数，其计算方式如下：
 X = Innodb_buffer_pool_pages_total – Innodb_buffer_pool_pages_free –Innodb_buffer_pool_pages_data
- Innodb_buffer_pool_pages_total
 缓冲池中的总页数。
- Innodb_buffer_pool_read_ahead_rnd
 InnoDB 扫描大块数据时发生随机读头的数量。
- Innodb_buffer_pool_read_ahead_seq
 顺序全表扫描时发生的顺序读取表头的数量。
- Innodb_buffer_pool_read_requests
 逻辑读请求的次数。
- Innodb_buffer_pool_reads
 直接从磁盘中逻辑读取（而不是从缓冲池读）的次数。

- `Innodb_buffer_pool_wait_free`
 如果缓冲池繁忙且没有空页，InnoDB 可能需要等待页面刷新。该值表示等待的次数。如果这个值增加且始终大于 0，可能是缓冲池过小或磁盘访问出问题。
- `Innodb_buffer_pool_write_requests`
 写入 InnoDB 缓冲池的次数。

363 所有这些选项显示的都是数字信息，可以在 MySQL 管理器中建立描述这些信息的自定义图表。

监控表空间

如果 InnoDB 可以在运行缓慢时扩展表空间，那么 InnoDB 的表空间基本可以自给自足。配置 `innodb_data_file_path` 变量的 `autoextend` 选项，允许 InnoDB 表空间自动扩展。例如，MySQL 安装时默认的共享空间大小为 10MB，且可以自动扩展为多个文件。

```
--innodb_data_file_path=ibdata1:10M:autoextend
```

详见 MySQL 在线参考手册的“InnoDB 配置”章节。

使用 `SHOW ENGINE STATUS INNODB` 命令查看当前的表空间配置信息。还可以通过打开 InnoDB 表空间监控器查看表空间的详细信息（详见在线 MySQL 参考手册中的“使用表空间监控器”）。

使用INFORMATION_SCHEMA表

如果你安装的 MySQL 版本含有 InnoDB 存储引擎插件（MySQL 5.1 或更高的版本可用），还可以在 *INFORMATION_SCHEMA* 数据库中访问 7 个特殊表。



必须单独安装 *INFORMATION_SCHEMA* 表。请参阅 InnoDB 插件文档以获取更多细节信息（*http://www.innodb.com/products/innodb_plugin/plugin-documentation/*）。

从技术上来讲，这些表并不是真正意义上的表，其数据不是存储在磁盘上的，而是在查询表时生成的。这些表提供了另一种监控 InnoDB 的方法，并为管理员提供性能信息。这些表用于监控压缩、事务和锁，我们将依次简单介绍。

- *INNODB_CMP*
 显示压缩表的详细信息和统计信息。
- *INNODB_CMP_RESET*
 与 *INNODB_CMP* 显示相同的信息，但是其有个特性就是：在查询表时将重置统计

信息，这使你可以定期（如每小时、每天等）跟踪统计信息。

- *INNODB_CMPMEM*
 显示在缓冲池中使用压缩的详细信息和统计信息。
- *INNODB_CMPMEM_RESET*
 与 *INNODB_CMPMEM* 显示相同的信息，但是其有个特性就是：在查询表时将重置统计信息，这使你可以定期（如每小时、每天等）跟踪统计信息。
- *INNODB_TRX*
 显示所有事务的详细信息和统计信息，包括事务状态和当前正在运行的查询信息。
- *INNODB_LOCKS*
 显示事务请求的锁的详细信息和统计信息。描述每个锁的状态、模式、类型等信息。
- *INNODB_LOCK_WAITS*
 显示被阻塞的事务请求的锁的详细信息和统计信息，描述每个锁的状态、模式、类型和阻塞事务。



InnoDB 插件文档中描述了每个表的完整信息，其中包括表的列信息和如何使用每个表的实例。

可以使用压缩表监控表的压缩信息，其中包括以下细节信息：页大小、使用哪些页、压缩时间和解压时间等。如果使用了压缩并希望压缩带来的开销不影响数据库服务器性能，那么这些信息将是重要监控对象。

可以使用事务和锁定表来监控事务。这是一个非常有用的工具，可以保证事务数据库顺利运行，更重要的是，通过它可以精确地确定各个事务的状态，以及哪些事务受阻，哪些处于锁定状态。这些信息对于诊断复杂的事务问题（如死锁或低性能）也是很重要的。

其他需要考虑的参数

还有很多其他参数用于监控和优化 InnoDB 存储引擎。前面我们只讨论了其中的一部分，主要介绍有关监控子系统和提高性能的部分。当然，还有一些其他参数可能需要考虑。

某些情况下，可以通过调节 `innodb_thread_concurrency` 选项提高线程性能。该参数默认值是 0，一般情况下，该值是足够的。但如果在多处理器及多个独立磁盘的服务器上运行 MySQL（并频繁使用 InnoDB），那么将此值设置为处理器个数加上独立磁盘数的和，系统性能会提高，确保 InnoDB 使用足够的线程最大化并发操作。如果该值大于系统所能支持的值，则起不到任何作用或作用不大——如果没有任何可用的线程，这样的设置将无法达到最大值。

如果系统频繁或甚至定期被关闭（例如，在 Linux 系统启动时运行 MySQL），可能关闭 InnoDB 需要花费很长时间。幸运的是，可以通过设置 innodb_fast_shutdown 选项来快速关闭 InnoDB。这不会影响数据的完整性，也不会导致内存（缓冲）管理损失。快速关闭 InnoDB 只是简单地跳过清除内部缓存和合并插入缓冲这些潜在的高成本操作，仍然执行一个可控的关闭过程，并在磁盘上存储缓冲池。

MySQL 的早期发行版本有并发控制和锁问题。对于这些早期版本，可以通过设置 innodb_lock_wait_timeout 变量来控制 InnoDB 如何处理死锁。该变量有全局和会话范围，控制 InnoDB 允许事务在终止之前等待行锁的等待时间，默认值是 50 秒。如果有很多锁等待超时，可以增加该变量值，缓和一些并发问题。

如果你正在导入大量数据，确保将传入的数据文件按主键的顺序排序，这样可以改善装载时间。此外，将 AUTOCOMMIT 设为 0 关闭自动提交，保证整个装载只提交一次。还可以通过关闭外键和唯一约束来改善批量装载。



记住，应该非常谨慎地优化 InnoDB。InnoDB 优化过程中有很多东西需要调整，很容易出错。一定要遵循以下原则：一次只调整一个参数（仅为一个目的），然后不停地测试。

366

小结

本章研究如何监控和提高 MySQL 服务器中的存储引擎性能。我们已经详细讨论了两种最流行的存储引擎，下一章将讨论更高级的主题：监控和提高复制性能。

Joel 将鼠标停留在发送键上。他刚刚完成了一份 InnoDB 监控数据报告，并在邮件里写了一些建议，但是他不确定是否应该在老板还没要之前就发邮件。他耸了耸肩，心想反正没什么坏处，就点击了发送邮件。

大约两分钟后，系统弹出一个消息框提示有新邮件。Joel 打开邮件，是老板发过来的。

“Joel，InnoDB 的事情做得不错嘛，我希望你能够跟开发者和 IT 人员一起开个会，大家坐下来谈谈你的建议。你来安排吧，我星期一来公司。”

“好的。”Joel 说，感觉肩上多了份责任感，这可是他上班以来的第一次会议。他有些紧张，所以决定出去散散步，然后再把相关议程发送给需要参加会议的人。“肯定不会比我的论文答辩更糟糕吧。”

复制监控

Joel 花了点时间登录西雅图的复制 Slave，确定复制仍在运行。

一个熟悉的声音从门口传来："西雅图那边怎么样了，Joel，你在跟进这件事情吗？"

"正在处理中，先生。我需要找出复制的配置信息并监控问题所在。"Joel 心想，还要阅读更多关于复制监控的知识。

"好的，Joel。继续工作吧，午饭后我再来检查。"

当老板离开后，Joel 看了看表，大约还有一个小时时间考虑如何监控复制。

Joel 深深地叹了一口气，再一次打开自己喜爱的 MySQL 书，了解更多有关 MySQL 监控的知识。"我没想到复制本身会带来这么多问题。"他喃喃地说。

你已经知道服务器什么时候正常运行（以及什么时候不正常），那么如何知道复制的运行情况呢？可能会正常运行，但是如何知道呢？

本章将讨论高级监控，重点介绍监控和提高复制性能。

开始

有两点能够影响复制拓扑性能，必须优化它们，以免影响复制性能。

首先，确保网络有足够的带宽去处理复制数据。正如我们所讨论的，Master 产生更新副本，然后通过网络将副本发送给 Slave。如果网络链接很慢或者网络资源竞争很激烈，那么

368 复制数据也将运行得很慢。我们将介绍一些优化网络和复制的方法，以最大限度地利用当前的网络环境。

其次，确保被复制的数据库是经过优化的。这一点很重要，因为 Master 上的数据库中的任何无效率事例都将被同步到 Slave 上，从而导致 Slave 的数据库性能也不高，对于索引和规范化而言尤其如此。然而，优化数据库只是确保复制优化的一部分，还必须确保查询语句的优化。例如，一个优化不佳的查询既在 Master 上运行缓慢，也会在 Slave 上运行不佳。

网络运行良好，且数据库和查询都经过优化了，就可以专注于配置服务器以获得最优性能了。

安装服务器

为复制拓扑创建最佳平台还有一个很重要的原因，就是确保服务器的配置性能最优化。性能不好的复制拓扑往往是由于性能不佳的服务器导致的。请确保服务器操作系统拥有足够内存，并且存储设备和存储引擎都已经被优化。

有人建议 Slave 使用低性能的机器，因为 Slave 上的运行负载较少（一般来说，Slave 只处理 SELECT 查询语句，而 Master 需要处理更新数据）。然而，这是不正确的，对于典型的单 Master 和单 Slave 环境，所有的数据库都需要复制，两个机器的负载相同，但是 Slave 使用单线程执行事件，而 Master 使用多线程执行事件，所以即使负载相同，Slave 处理和执行事件所花的时间可能更多。

对此最好的办法可能是：最好也用复制做故障转移。如果 Slave 比 Master 慢，并且如果 Master 出现故障就必须故障转移，那么性能即将提升的 Slave 应该与崩溃的 Master 拥有相同的性能。

包容性和排他性复制

可以将复制配置成复制所有的数据（默认配置），或者只记录 Master 上的某些数据或者忽略某些数据，从而限制了哪些写入二进制日志，哪些被复制。或者还可以配置 Slave，使其只对某些数据进行操作。使用包容性或排他性复制（或同时用两种复制）有助于解决复杂的负载均衡或系统扩展问题，使复制更强大、更灵活。这个过程又称为过滤数据，其中包容性和排他性要求形成了过滤规则。

369 在 Master 上使用 `--binlog-do-db` 启动选项指定哪个数据库的事件需要写入二进制日志。可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。

还可以使用 --binlog-ignore-db 启动选项指定数据库需要忽略的选项，可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。该选项告诉 Master 列表上的这些数据库事件不需要记录。



可以同时使用 --binlog-do-db 和 --binlog-ignore-db 选项，但是一定需要在诊断数据复制问题（例如 Slave 数据丢失）时检查这些变量的值。

另外，使用 -binlog-do-db 或 -binlog-ignore-db 选项会过滤即将写入二进制日志的数据。这严重限制了 PITR 的使用，因为只有写入二进制日志的数据才能恢复。

通过选项来控制哪些数据需要复制到 Slave。还有几个选项，如 Master 上的 binlog 选项，限制表级别的选项，甚至还有做转换（重命名）的命令选项。



在 Slave 上执行包容性或排他性复制或许无法提高拓扑复制的性能。尽管 Slave 上的数据较少，Master 还是需要传输相同的数据量，而且如果包容性和排他性列表很复杂，那么在 Slave 上进行过滤也没什么好处。如果你担心通过网络传输过多的数据，最好在 Master 上执行过滤操作。

370

使用 --replicate-do-db 启动选项指定 Slave 数据库上的哪些事件可以从中继日志中读取然后执行。可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。该选项告诉 Slave 只执行指定数据库上的某些事件。

使用 --replicate-ignore-db 启动选项指定哪些数据库事件将被忽略。可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。该选项告诉 Slave 忽略指定数据库上的所有事件。



Slave 上的复制选项根据使用格式不同而作用不同。对于基于语句的复制而言，这一点尤为重要，稍有不慎就可能导致数据丢失。例如，如果你使用基于语句的复制，且使用 --replicate-do-db 选项，那么 Slave 只对 USE *<db>* 命令后的那些语句的事件起作用。如果你在不改变数据库的情况下向不同的数据库发送语句，该语句将会被忽略。想要了解更多信息，请参看 MySQL 在线参考文档。

可以在 Slave 上执行表级别的包容性和排他性复制。使用 --replicate-do-table 和 --replicate-ignore-table 选项执行或忽略那些特定表的事件。在有些包含敏感数据的表中（这些表用于管理或某些特殊用途），当需要禁止应用程序访问的时候，上述命令是非常有用的。例如，如果你有一个应用程序，包含供应商的产品价格信息（你付出的钱），可能希望在雇佣或外包销售服务时隐藏这些信息。在这种情况下，不需要单独建立一个有关承包商的特殊应用，可以通过部署现有的应用程序，使用 Slave 复制除有敏感数据

的表外的所有信息。

上面介绍的两个选项还可以使用通配符形式。replicate-wild-do-table和replicate-wild-ignore-table与前面两个选项功能相同，但是还支持通配符。例如，--replicate-wild-do-table=tbl%执行任何以“tbl”开头（如*tbl*、*tbl1*、*tbl_test*）的所有表事件。这些通配符在Slave端过滤，有利于解决复杂复制问题。

还可以在Slave上使用转换选项，用于重命名或改变表操作对应的数据库。该选项仅适用于表，格式为--replicate-rewrite-db="<from>-><to>"（必须加引号）。该选项只改变和表事件对应的数据库名，不影响CREATE DATABASE, ALTER DATABASE等命令。该选项只影响指定数据库的事件（或者为基于语句的复制重定向默认数据库）。可以多次使用该选项以改变多个数据库的名称。



虽然--replicate-same-server-id不是严格意义上的包容性或排他性选项，但可以防止循环复制中产生无限循环。如果将其值设置为0，Slave将跳过相同server_id的事件；如果将其值设置为1，Slave将执行所有事件。

371

复制线程

在谈论Master和Slave的监控问题之前，首先看看复制中的线程问题。这里从监控和诊断问题的角度再说明一下。

控制复制有三个线程，每个线程执行一个特定的角色。在Master上，每个已连接的Slave都有一个线程，称为Binlog Dump线程。该线程负责将binlog事件传送给已连接的Slave。Slave上有两个线程，即Slave IO线程和Slave SQL线程。I/O线程负责读取Master传过来的binlog事件，然后将这些事件写入Slave的中继日志。SQL线程负责读取并在之后执行中继日志中的事件然后执行。

使用SHOW PROCESS LIST命令可以监控Binlog Dump线程当前的状态：

```
mysql> SHOW PROCESSLIST \G
*************************** 1. row ***************************
。
   User: rpl
   Host: localhost:54197
     db: NULL
Command: Binlog Dump
   Time: 25
  State: Master has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

请注意 State 列，描述 Master 对二进制日志和 Slave 做了哪些操作。上面的实例显示了一个运行良好的复制拓扑产生的典型结果。其显示结果中还包含以下字段：

- Id
 显示连接 ID。
- User
 显示运行该语句的用户。
- Host
 语句来源的主机。
- db
 指定的默认数据库，如果未指定，则显示 NULL，表明没有指定默认数据库。
- Command
 该线程运行的命令类型。参见 MySQL 在线参考手册以获取更多信息。
- Time
 线程处于报告状态的时间（以秒为单位）。
- State
 当前动作或状态（如等待）的描述信息。通常是描述性文本信息。
- Info
 线程正在执行的语句信息。NULL 表明无语句执行，例如复制线程处于等待状态时该字段为 NULL。

372

还可以在 Slave 上查看线程状态。使用 SHOW PROCESSLIST 命令监控 I/O 和 SQL 线程。

```
mysql> SHOW PROCESSLIST \G
*************************** 1. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 127
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 3
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 10
  State: Slave has read all relay log; waiting for the slave I/O thread to
         update it
```

```
    Info: NULL
```

同样，State列包含了最重要的信息。如果Slave上的复制出了问题，确保在Slave上执行SHOW PROCESSLIST命令，并留意I/O和SQL线程的状态。在上面的例子中，我们看到了Slave的正常状态，即等待来自Master的信息（I/O线程）并且执行完毕中继日志中的所有事件（SQL线程）。



在故障排除时，最好使用SHOW PROCESSLIST命令参看复制状态。

监控Master

监控Master的方法很多。使用SHOW命令可以查看状态信息和状态变量，或者使用MySQL管理器。主要的SQL命令包括：SHOW MASTER STATUS、SHOW BINARY和SHOW BINLOG EVENTS。

373 本节我们将看看哪些命令可用于监控Master，然后简单总结一下可监控的状态变量，这些状态变量可以通过SHOW STATUS命令或使用MySQL管理器创建的自定义图表获得。

Master的监控命令

SHOW MASTER STATUS命令显示Master的二进制日志的相关信息，其中包括当前binlog文件名及其偏移位置。连接Slave时这些信息很重要，这一点上一章已经讲过。

还有日志约束的相关信息。例10-1显示了SHOW MASTER STATUS命令的结果。

例10-1：SHOW MASTER STATUS命令

```
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000002 | 156058362 | Inventory    | Vendor_sales     |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

显示结果包含以下字段：

- File
 列出当前binlog文件的名称。
- Position
 列出二进制日志的当前位置（即下一次写入的位置）。

- Binlog_Do_DB
 列出 --binlog-do-db 启动选项指定的所有数据库。
- Binlog_Ignore_DB
 列出 --binlog-ignore-db 启动选项指定的所有数据库。

SHOW BINARY LOGS 命令（或 SHOW MASTER LOGS 命令）列出了 Master 上可用的二进制文件及其文件大小（以字节为单位）。该命令有利于比较 Master 和 Slave 的信息，即 Slave 目前正在读取哪个 Master 上的二进制日志。例 10-2 显示了 SHOW BINARY LOGS 命令的结果信息。

例10-2：SHOW MASTER LOGS命令 374

```
mysql> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| master-bin.000001 | 103648205 |
| master-bin.000002 | 2045693   |
| master-bin.000003 | 1022910   |
| master-bin.000004 | 3068436   |
+------------------+-----------+
4 rows in set (0.00 sec)
```



在 Master 上运行 FLUSH LOGS 命令可以轮换二进制日志。该命令关闭并重新打开所有日志，然后以递增的文件扩展名打开一个新日志文件。定期刷新日志，以管理日益增长的日志，同时还有助于诊断复制问题。

使用 SHOW BINLOG EVENTS 命令可以显示二进制日志中的事件，该命令的语法如下：

```
SHOW BINLOG EVENTS [IN <log>] [FROM <pos>] [LIMIT [<offset>,] <rows>]
```

使用这个命令时需要注意，因为它会产生大量数据。该命令最好的用处是：将 Master 上的事件与 Slave 中继日志中的事件进行对比。例 10-3 显示了典型复制配置中的 binlog 事件。

例10-3：SHOW BINLOG EVENTS 命令（基于语句复制）

```
mysql> SHOW BINLOG EVENTS IN 'master-bin.000001' FROM 2571 LIMIT 4 \G
*************************** 1. row ***************************
   Log_name: master-bin.000001
        Pos: 2571
 Event_type: Query
  Server_id: 1
End_log_pos: 2968
       Info: use 'employees'; CREATE TABLE salaries (
    emp_no      INT             NOT NULL,
```

```
    salary      INT             NOT NULL,
    from_date   DATE            NOT NULL,
    to_date     DATE            NOT NULL,
    KEY         (emp_no),
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no)
    ON DELETE CASCADE,
    PRIMARY KEY (emp_no, from_date)
)
*************************** 2. row ***************************
   Log_name: master-bin.000001
        Pos: 2968
 Event_type: Query
  Server_id: 1
End_log_pos: 3041
       Info: BEGIN
*************************** 3. row ***************************
   Log_name: master-bin.000001
        Pos: 3041
 Event_type: Query
  Server_id: 1
End_log_pos: 3348
       Info: use 'employees'; INSERT INTO 'departments' VALUES
       ('d001','Marketing'),('d002','Finance'),('d003','Human Resources'),
       ('d004','Production'),('d005','Development'),('d006','Quality
       Management'),('d007','Sales'),('d008','Research'),('d009',
      'Customer Service')
*************************** 4. row ***************************
   Log_name: master-bin.000001
        Pos: 3348
 Event_type: Xid
  Server_id: 1
End_log_pos: 3375
       Info: COMMIT /* xid=17 */
4 rows in set (0.01 sec)
```

375

本例使用基于语句的复制。如果使用基于行的复制，则显示不同的binlog事件，如例10-4所示。

例10-4：SHOW BINLOG EVENTS命令（基于行的复制）

```
mysql> SHOW BINLOG EVENTS IN 'master-bin.000001' FROM 2571 LIMIT 4 \G
*************************** 1. row ***************************
   Log_name: master-bin.000001
        Pos: 2571
 Event_type: Query
  Server_id: 1
End_log_pos: 2968
       Info: use 'employees'; CREATE TABLE salaries (
    emp_no      INT             NOT NULL,
    salary      INT             NOT NULL,
```

```
    from_date   DATE            NOT NULL,
    to_date     DATE            NOT NULL,
    KEY         (emp_no),
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,
    PRIMARY KEY (emp_no, from_date)
)
*************************** 2. row ***************************
   Log_name: master-bin.000001
        Pos: 2968
 Event_type: Query
  Server_id: 1
End_log_pos: 3041
       Info: BEGIN
*************************** 3. row ***************************
   Log_name: master-bin.000001
        Pos: 3041
 Event_type: Table_map
  Server_id: 1
End_log_pos: 3101
       Info: table_id: 15 (employees.departments)
*************************** 4. row ***************************
   Log_name: master-bin.000001
        Pos: 3101
 Event_type: Write_rows
  Server_id: 1
End_log_pos: 3292
       Info: table_id: 15 flags: STMT_END_F
4 rows in set (0.01 sec)
```

376

请注意基于行格式的二进制日志信息很少。但是当在诊断数据崩溃或间歇性错误这些复杂问题时，有时候切换成基于语句的行格式是很有用的。例如，这样将有利于查看写入Master二进制日志的信息内容，然后将其与Slave中继日志的读取结果进行比较。如果不同，那么基于语句的格式比基于行的格式更方便查找。第2章详细介绍了二进制日志的格式，以及各种格式之间的优缺点比较。

Master的状态变量

监控Master的状态变量较少，只有计数器指明Master上的命令执行的次数。

- `Com_change_master`
 显示`CHANGE MASTER`命令执行的次数。如果该值变化得很频繁，或者该值比服务器数目与Slave计划重启次数的乘积高很多，有可能是附加的Slave重启过于频繁，这表明连接不稳定。
- `Com_show_master_status`
 显示`SHOW MASTER STATUS`命令执行的次数。与`Com_change_master`相同，该值如

果很高，表明 Slave 的重连接请求的次数不正常。

监控Slave

监控 Slave 的方法很多，可以使用 SHOW 命令查看其状态信息和状态变量，或者使用 MySQL 管理器。SQL 命令主要有 SHOW SLAVE STATUS、SHOW BINARY LOGS 和 SHOW BINLOG EVENTS。

本节将讨论用于监控 Slave 的 SQL 命令及可用的状态变量，可以使用 SHOW STATUA 命令，或通过 MySQL 管理器创建的自定义图标监控这些状态变量。我们将在后面的“使用 MySQL 管理器监控复制”中介绍 MySQL 管理器。

377

Slave的监控命令

SHOW SLAVE STATUS 命令显示以下信息：Slave 的二进制日志、Slave 到 Master 的连接和复制活动，包括当前 binlog 文件的文件名及其偏移位置。这些信息在诊断 Slave 性能时非常重要，就像我们在前面的章节看到的一样。例 10-5 显示了在 MySQL 5.5 上执行的 SHOW SLAVE STATUS 命令。

例10-5：SHOW SLAVE STATUS命令

```
mysql> SHOW SLAVE STATUS \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: localhost
                  Master_User: rpl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000002
          Read_Master_Log_Pos: 39016226
               Relay_Log_File: relay-bin.000004
                Relay_Log_Pos: 9353715
        Relay_Master_Log_File: mysql-bin.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 25263417
```

```
              Relay_Log_Space: 39016668
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 66
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids:
             Master_Server_Id: 1
1 row in set (0.00 sec)
```


以上例子显示了大量信息。该命令是有关复制的最重要命令，最好能仔细研究上面例子中的各项。我们从管理员的角度解释这些信息，而不是以列表的形式逐项列出。也就是说，这些信息通常是有明确的目的意义的。因此，我们将这些信息分组以方便参看。这些分类包括：Master 连接信息、Slave 性能、日志信息、过滤、日志性能和出错条件。

最重要的是第一列，显示了当前 I/O 线程的状态，包括连接 Master、等待来自 Master 的事件、重新连接 Master 等。

还有 Master 连接的相关信息，包括当前 Master 的主机名、连接的用户账号，以及用于连接 Master 的 Slave 端口。最后是 SSL 连接信息（如果使用了 SSL 连接）。

下一类是关于 Master 二进制日志和 Slave 中继日志的信息，包括文件名和位置信息。诊断复制问题时，必须留意这些值，尤其是 `Relay_Master_Log_File` 的值。该值表明了最近执行的中继日志事件所在的 Master 日志文件名。

复制过滤配置列举了所有 Slave 端的复制过滤器。如果不知道过滤器的配置，可以检查这个配置。

另外，还有最后一个错误号、Slave 文本及 I/O 线程和 SQL 线程信息。除了 Slave 线程的状态值以外，系统出现错误时常常需要检查这些信息。当 Slave 遇到错误时，在检查错误日志之前，最好先查看这些信息，因为这些信息最及时，通常可以告诉你出错的原因。

列表中还包含 Slave 的配置信息，其中包括跳跃计数器的设置和 `until` 条件。参看 MySQL 在线参考手册以获取更多该方面的信息。

列表底部附近的信息是当前错误信息，包括 Slave 的 I/O 和 SQL 线程错误信息。这些值应该保持在 0，这样 Slave 才能正常运行。

下面我们将详细讨论一些更重要的有关性能的列：

379

- `Connect_Retry`
 两次重连接的时间间隔（以秒计算），该值应该较小，但是如果 Slave 连接 Master 时出现问题，可以将该值设置得较大。
- `Exec_Master_Log_Pos`
 显示 Master 二进制日志中最后执行的事件位置。
- `Relay_Log_Space`
 所有中继日志文件的总大小。根据事件运行过程中的磁盘空间状况，确定是否需要清除中继日志。
- `Seconds_Behind_Master`
 事件执行和事件写入 Master 二进制日志之间的间隔时间（以秒计算）。该值过高表明有重大复制滞后。下面即将讨论复制滞后。



当复制由于网络错误、Master 的心跳丢失等原因停止时，`Seconds_Behind_Master` 的值将过时，只有在复制运行时才有效。

如果 Slave 启用了二进制日志，使用 `SHOW BINARY LOGS` 命令可以查看 Slave 上可用的 binlog 文件列表及其大小（单位为字节）。例 10-6 显示了 `SHOW BINARY LOGS` 命令的结果信息。

例 10-6：Slave SHOW BINARY LOGS 命令

```
mysql> SHOW BINARY LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| slave-bin.000001 |   5151604 |
| slave-bin.000002 |   1030108 |
| slave-bin.000003 |   1030044 |
+------------------+-----------+
3 rows in set (0.00 sec)
```



使用 FLUSH LOGS 命令可以轮换 Slave 上的中继日志。

如果启用了 Slave 上的二进制日志，还可以使用 SHOW BINLOG EVENTS 命令显示 Slave 上的二进制日志事件。在 Slave 上显示事件与在 Master 上显示事件的不同之处在于：前者可以在 SHOW BINARY LOGS 命令的输出结果中指定 Slave 上的 binlog 的文件名。例 10-7 显示了一个典型复制配置中的 binlog 事件。

例 10-7：SHOW BINLOG EVENTS 命令 380

```
mysql> SHOW BINLOG EVENTS IN 'slave-bin.000001' FROM 2701 LIMIT 2 \G
*************************** 1. row ***************************
   Log_name: slave-bin.000001
        Pos: 2701
 Event_type: Query
  Server_id: 1
End_log_pos: 3098
       Info: use 'employees'; CREATE TABLE salaries (
    emp_no      INT             NOT NULL,
    salary      INT             NOT NULL,
    from_date   DATE            NOT NULL,
    to_date     DATE            NOT NULL,
    KEY         (emp_no),
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,
    PRIMARY KEY (emp_no, from_date)
)
*************************** 2. row ***************************
   Log_name: slave-bin.000001
        Pos: 3098
 Event_type: Query
  Server_id: 1
End_log_pos: 3405
       Info: use 'employees'; INSERT INTO 'departments' VALUES
    ('d001','Marketing'),('d002','Finance'),
    ('d003','Human Resources'),('d004','Production'),
    ('d005','Development'),('d006','Quality Management'),
    ('d007','Sales'),('d008','Research'),
    ('d009','Customer Service')
2 rows in set (0.01 sec)
```



在 MySQL 5.5 及其以后的版本中，还可以使用 SHOW RELAYLOG EVENTS 命令检查 Slave 的中继日志。

Slave的状态变量

只有少数几个状态变量用于监控 Slave。其中包括计数器，指明 Master 上 Slave 相关的命令的执行次数，以及关键 Slave 操作的统计数据。这里列出的前四个变量是 Slave 相

关的命令的计数器。这些值应该与 Slave 的维护频率相对应。如果它们不对应，可能需要调查以下两种情况：拓扑结构上 Slave 数量过多，或者某个 Slave 重启次数过于频繁。

381

- `Com_show_slave_hosts`
 `SHOW SLAVE HOSTS` 命令执行的次数。
- `Com_show_slave_status`
 `SHOW SLAVE STATUS` 命令执行的次数。
- `Com_slave_start`
 `SLAVE START` 命令执行的次数。
- `Com_slave_stop`
 `SLAVE STOP` 命令执行的次数。
- `Slave_heartbeat_period`
 当前 Master 的心跳检测的间隔时间的配置信息。
- `Slave_open_temp_tables`
 Slave 的 SQL 线程使用的临时表的数量。该值如果过高，则说明 Slave 负载过重。
- `Slave_received_heartbeats`
 从 Master 回复的心跳数，该值应该与心跳检测间隔时间一致，因为 Slave 的重启动会在心跳间隔时被断开。
- `Slave_retried_transactions`
 Slave 启动后 SQL 线程重试事务的次数。
- `Slave_running`
 如果 Slave 连接到 Master，而且 I/O 和 SQL 线程无错误地执行，则该值为 `ON`。

使用MySQL管理器监控复制

你已经知道了如何使用 MySQL 监控器监控网络流量和存储引擎。它还可以监控复制拓扑结构中的 Master 和 Slave。可以在复制状态栏中查看基本的复制信息。然而，要获得这些信息，需要使用 `--report_host` 启动选项启动 Slave，并为每个 Slave 提供唯一的名称。

图 10-1 显示了在 Master 上运行的 MySQL 管理器，该 Master 连接了一个 Slave。如果 Slave 连接时没有选择 `--report_host` 选项，那么列表中不会出现该 Slave。

如果你在 Slave 上运行 MySQL 管理器，则只能看到 Slave 的信息。图 10-2 显示了 Slave 上运行的 MySQL 管理器。

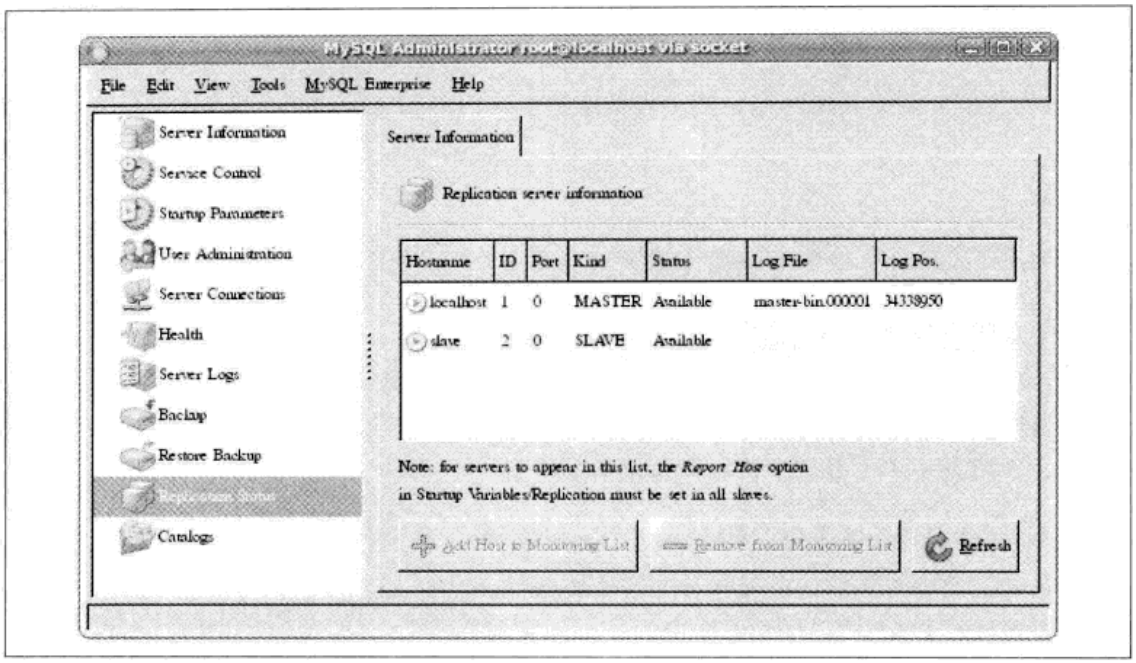


图 10-1：在Master上运行的MySQL 管理器

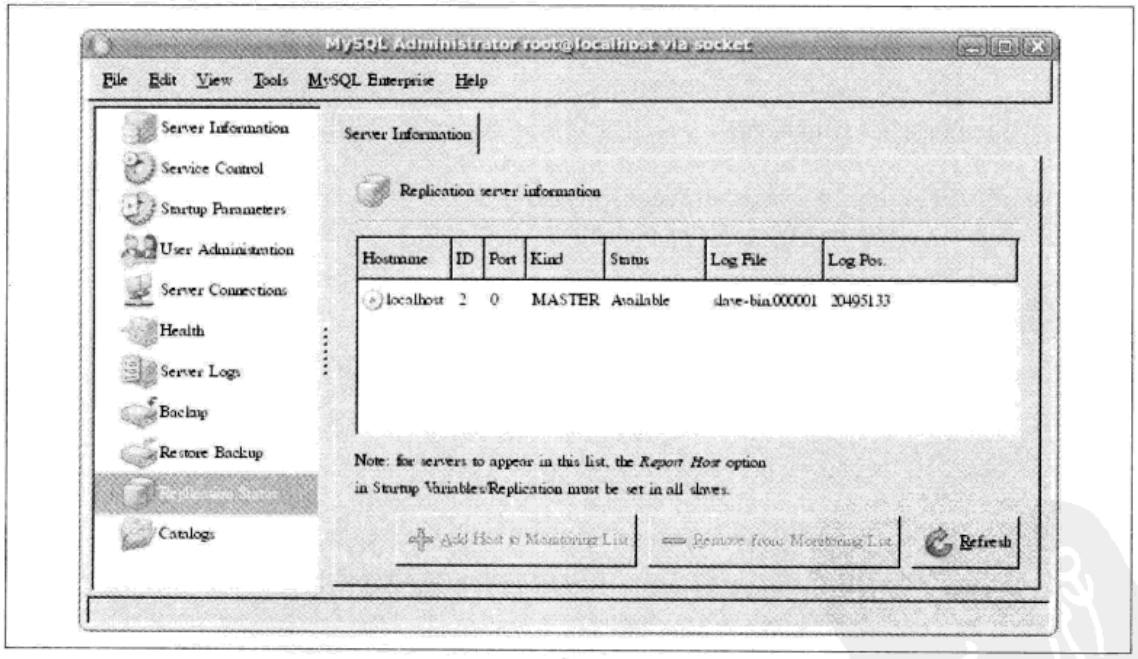


图 10-2：在Slave上运行的MySQL 管理器

图 10-1 和图 10-2 中显示的信息包括主机名、服务器 ID、端口、类型（Master 或 Slave）、一般状态、日志文件（binlog 文件名）和当前日志位置。图 10-1 显示列有所有已连接的 Slave 的复制拓扑结构。该报告可以使你一眼看清服务器的状态。

其他需要考虑的项

本节讨论监控复制的其他问题，其中包括特殊网络因素和监控滞后（复制延时）。

网络

如果网络带宽有限、带宽竞争激烈或者连接缓慢，可以使用压缩提高复制性能。使用 slave_compressed_protocol 变量配置压缩信息。

如果网络带宽不是问题，但是需要加密 Master 到 Slave 的数据，这时可以使用 SSL 连接。使用 CHANGE MASTER 命令配置 SSL 连接。参看 MySQL 在线文档中的“使用 SSL 设置复制”，了解更多关于复制中使用 SSL 连接的信息。

还有一个网络配置需要考虑，即 Master 心跳的使用。前面你已经看到了 SHOW SLAVE STATUS 命令的结果信息。心跳机制是指自动检测 Master 与 Slave 之间的连接状态，可以检测到毫秒级别的连接。复制方案中的 Master 心跳机制，Slave 必须与 Master 同步，并且（几乎）没有延迟。心跳机制可以在 Slave 上的复制停止之前，确保延迟在一定时间内被诊断出来。

可以使用 CHANGE MASTER 命令中的参数与 master_heartbeat_period=*<value>* 设置（在 MySQL 5.4.4 版本中加入的）配置 Master 的心跳，master_heartbeat_period 参数值是心跳执行的间隔时间（以秒计算）。可以使用以下命令监控心跳的状态：

```
SHOW STATUS like 'slave_heartbeat period'
SHOW STATUS like 'slave_received_heartbeats'
```

监控和管理Slave滞后

大规模的数据更新、Slave 负担过重或其他严重的网络性能事件都可以导致 Slave 滞后于 Master。当这种滞后发生时，Slave 将无法快速处理中继日志中的事件，从而无法与 Master 的更新保持一致。

384 使用 SHOW SLAVE STATUS 命令查看，Seconds_Behind_Master 表明 Slave 滞后于 Master。这个字段给出了 Slave 的 SQL 线程滞后于 I/O 线程的秒数，即 Slave 需要多久才能处理来自 Master 的事件。Slave 使用事件的时间戳计算该字段的值。当 Slave 上的 SQL 线程从 Master 读取事件时，线程将计算事件时间戳的差值。以下摘录显示了 Slave 滞后于 Master146 秒的情况。此时，Slave 滞后两分多钟。如果你的应用程序需要 Slave 提供及时的信息，则这种滞后将会引起问题。

```
mysql> SHOW SLAVE STATUS \G
```

```
...
           Seconds_Behind_Master: 146
...
```

SHOW PROCESSLIST 命令（在 Slave 上运行）也可以指明 Slave 延迟的时间。这里我们看到 SQL 线程滞后的时间秒数，该滞后时间是由最新复制事件的时间与 Slave 的实际运行该事件的时间之间的差计算得到的。例如，如果 Slave 在失去连接 30 分钟后再重新连接 Master，则 SHOW PROCESSLIST 的显示结果中的 Time 字段应为 1800 秒左右。下面的片段说明了这种情况。这里 Time 值越大说明 Slave 延迟越严重，将会产生过时数据。

```
mysql> SHOW PROCESSLIST \G
...
   Time: 1814
...
```

依赖于复制拓扑的设计方式，你可能需要复制数据以达到负载均衡。在这种情况下，一般使用多个 Slave，将一部分应用程序或用户的 SELECT 查询分配给 Slave 执行，这样将减少 Master 的负担。

Slave延迟的原因和预防措施

Slave 延迟令一些复制用户感到讨厌。Slave 延迟的主要原因是 Slave 是单线程的（事实上有两个线程，但是只有一个执行事件，这就是 Slave 滞后的主要原因）。例如，多核 Master 可以并行运行多个事务，且比在单线程运行事务（二进制日志中的事件）的 Slave 快。我们已经讨论过一些检测 Slave 延迟的方法。本节将讨论一些导致 Slave 延迟的常见原因及减少延迟的解决办法。

Slave 延迟的原因很多（例如网络延时）。可能是 Slave 的 I/O 线程推迟读取日志中的事件信息。最常见的原因是 Slave 是在单线程中执行所有的事务，而 Master 有很多线程可以并行执行事务。其他原因还有带有低效连接的长查询、磁盘读取的 I/O 限制、锁竞争和 InnoDB 线程同步启动等。 385

现在，了解了导致 Slave 延迟的原因，下面让我们学习如何减少 Slave 延迟：

- 组织数据
 通过标准化数据和使用数据分片来分布化数据的方法，可以提高性能。这有助于减少数据复制，但是就像第 8 章中提过的，实际上有些数据的复制（如 lookup 文本）可以提高系统性能。仅仅使用足够的标准化和数据分片来提高系统性能是远远不够的。现实中往往需要数据拥有者自己确定是否通过经验或试验来解决这个问题。
- 分而治之
 我们知道额外增加 Slave 去处理查询是提高性能的好办法，但是如果 Slave 执行的查

询非常多，向外扩展得不充分仍会导致 Slave 延迟。最坏的情况是所有的 Slave 都有延迟。为解决这个问题，考虑使用复制过滤器分割数据，在不同的 Slave 上复制不同的数据库。你仍可以使用向外扩展，但是在这种情况下，先使用中继 Slave 处理过滤的数据库组，然后再进行向外扩展。

- 识别并重构长连接的查询
 如果长时间运行的查询导致 Slave 滞后，考虑重构查询或操作或者操作或应用，发出较短的查询或更紧凑的事务。然而，如果你将该技术与复制过滤结合，就必须在发出跨复制过滤组的事务时留心，一旦你分割了一个长时间运行的查询，保证跨 Slave 的操作是原子操作（一个事务），就可能带来数据完整性问题。
- 负载均衡
 还可以使用负载均衡将查询重定向到不同的 Slave 中执行。这样可以减少每个 Slave 执行查询的时间，从而将有更多的计算时间去处理复制事件。
- 确保使用最新的硬件
 显然，具有最佳性能的硬件通常工作性能更好。最起码，应该确保 Slave 的硬件配置是最优的，至少要与 Master 一样强大。
- 减少锁竞争
 MyISAM 的表锁和 InnoDB 的行级锁可能导致 Slave 延迟。如果查询导致大量 MyISAM 或者 InnoDB 表被锁定，请考虑重构查询，尽量避免过多的锁。

386

小结

本章总结了监控 MySQL 的很多方法，并提供了监控 MySQL 服务器各个方面的基础。

现在你了解了基本的操作系统监控、数据库性能及 MySQL 监控和基准，这样就拥有可以成功优化服务器性能的工具和知识。

写复制问题报告的时候 Joel 笑了。他停下来看看门口，似乎觉得老板就要来了。

“Joel。”

Joel 吓了一跳，简直无法相信自己的预感。

“先生，我已经解决了复制问题。”他赶紧说。

"很好！那你有空时将详细资料发给我吧。"

“我还发现订单处理系统的一些有趣的事情。”看到 Summerson 先生的眉毛微微扬

起。Joel 继续说道："看起来我们设置的缓冲池大小不正确。我认为我可以就此做些改进。"

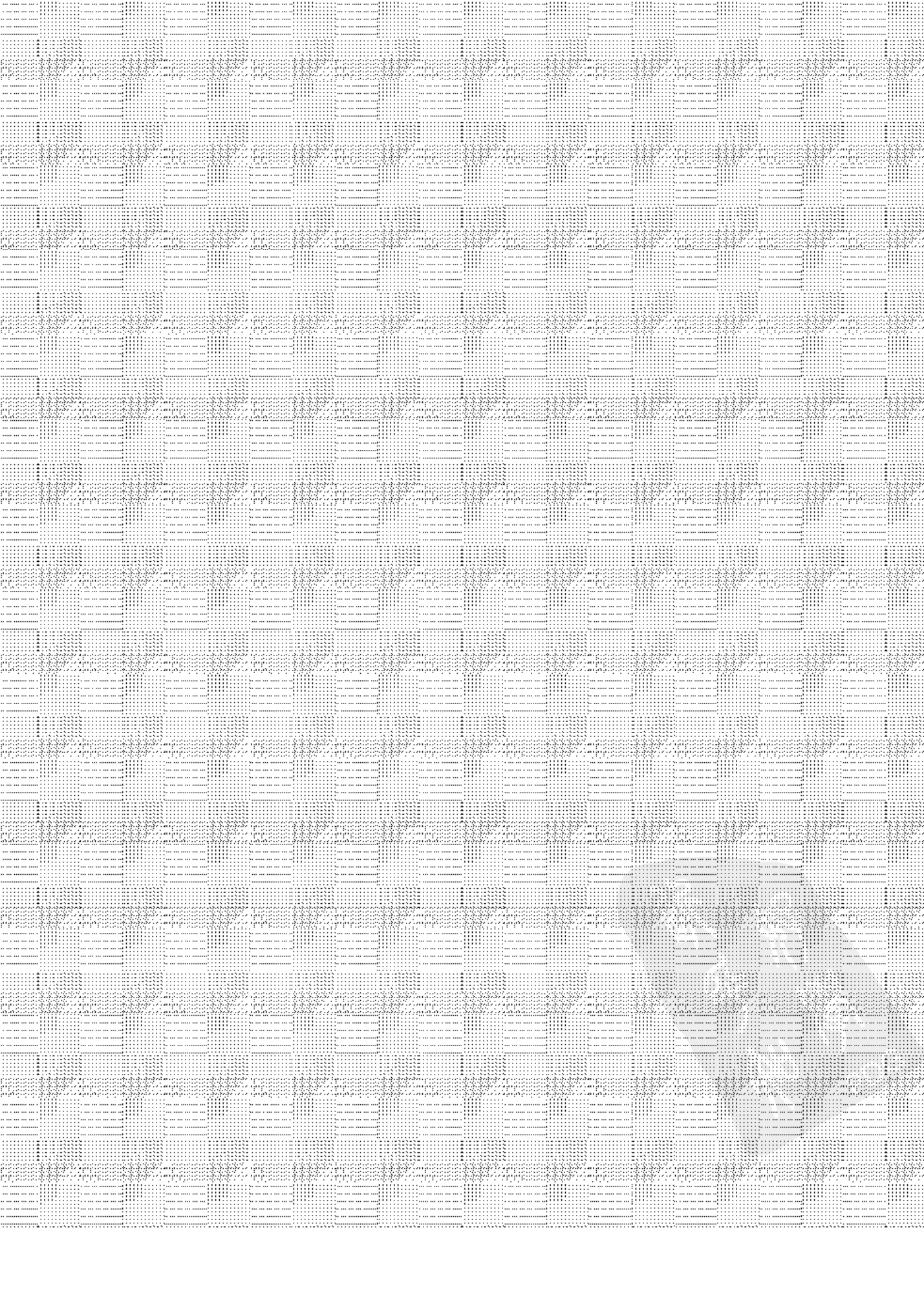
Summerson 先生说："又要监控？"

"是的，先生。我已经获得了 InnoDB 存储引擎的报告，我把这个也发给你吧。"

"做得好，确实很好。"

Joel 了解老板的这种表情。老板想了两次，这表明还有更多的工作。

令 Joel 惊讶的是，老板只是慢慢地走开了。Joel 心想："看来我终于难倒他了。"



第11章 387

复制的故障排除

邮件的标题只是简单地写着“修复西雅图的服务器”，Joel知道如此神秘的邮件只可能来自于Summerson先生，他快速阅读完邮件头信息后，确定该邮件来自于Summerson先生，Joel打开邮件开始阅读里面的内容。

“西雅图的服务器又出问题了，我认为问题出在数据复制上。请把这件事当做你的首要任务来解决。”

“好的。”Joel喃喃自语。由于他上周生成的监控报告没有显示异常，所以他确定上次检查时复制的设置是正确的。Joel不知道这个问题该如何下手，但是他知道在什么地方可以找到解决方法。“看来我需要阅读‘复制的故障排除’这一章节。”

正在这时，一个熟悉的身影出现在门口。Joel决定先发制人：“我正在解决这个问题。”听到这句话后，老板点头示意，然后继续往楼下走去。

如果拓扑结构是活动的且配置正确，MySQL的复制通常是无故障的，且很少需要调整。但是，也有出错误的时候。有时错误很明显，可能会有明确的证据用于展开调查，其他时候的情况和问题本身也很容易理解，但是某些复杂问题的原因却不是那么明显。幸运的是，如果你遵循一些简单的排除复制故障的准则和做法，就可以解决这些问题。

本章重点介绍如何通过技术来解决复制问题。首先了解导致复制问题的原因，然后讨论能够帮助排除故障的基本工具，最后总结一些预防和解决复制问题的策略。



MySQL集群的复制的故障排除与本章提出的流程一致。如果你不了解MySQL集群问题，请参看15章中的集群错误和启动问题的故障排除。 388

经验丰富的计算机用户知道计算系统容易偶尔出错。信息技术专家认为应该防止错误，确保为用户提供可靠的访问和数据，并将此作为他们的信条。然而，即使是妥善管理的系统也有可能出现问题。

MySQL 复制也不例外，特别是当 Slave 状态不是崩溃安全（crash-safe）时。也就是说，如果 Slave 上的 MySQL 实例崩溃，有可能 Slave 停止在一个未定义的状态。在最坏的情况下，中继日志或 *master.info* 文件可能已经损坏。

事实上，拓扑越复杂（包括负载和数据库的复杂程度）、拓扑节点上角色越多样化，系统越容易出现错误。这并不意味着复制不能够扩展，相反，复制很容易被扩展成大规模的复制拓扑结构。我们要说的是，当复制发生问题时，经常是由意外操作或配置的改变引起的。

什么导致错误发生

有很多情况可能导致复制故障发生。MySQL 的复制最容易出现数据问题，无论是数据损坏或复制流的意外中断。另外，系统崩溃导致的不安全及无控制的 MySQL 停机也可能导致复制重新启动的问题。

应该总是在为修复问题而改变某些东西之前，对数据进行备份。在某些情况下，备份将可能包含已损坏或丢失的数据，但是备份的好处仍然是有效的，特别是，无论你做什么，至少可以将数据恢复到错误阶段当时的状态。否则你会惊奇地发现，很容易将糟糕的事情变得更糟糕。

本节将通过描述 MySQL 复制中常见的错误来开始探索复制的故障排除。这些都是经常遇到的复制问题。虽然该列表并没有列出所有复制问题，但是它为你提供了可能出错的类型。我们也简短描述了每个问题可能的原因。

Master上的问题

虽然大多数错误出现在 Slave 上，但是本节将介绍 Master 上出现问题时的解决方案。当
389
发生错误时，管理员有时会自然地怀疑是 Slave 出现错误。应该在诊断复制问题时，同时看看 Master 和 Slave 的情况。

内存表在使用时Master崩溃

当 Master 重新启动时，内存表中的任何数据都会被清除（这对于内存存储引擎而言是正常的）。然而，如果使用了内存存储引擎的表（因此被称为内存表）正被复制，Slave 在服务器没有重启的情况下可能会含有过时数据。

幸运的是，当在重启之后第一次访问内存表时，一个特殊的删除事件被发送到 Slave 上，让 Slave 清除过时数据，从而达到数据同步。然而，数据被引用和复制事件被传送的时间间隙，可能导致 Slave 含有过时数据。为了避免该问题，请使用脚本先清除数据，然后在重启时使用 `init_file` 选项重新在 Master 上填充数据。

例如，如果你有一个内存表存储了一些频繁使用的数据，请创建如下所示的文件并用 `init_file` 选项引用它：

```
# Force slaves to purge data
DELETE FROM db1.mem_zip;
# Repopulate the data
INSERT INTO ...
```

第一个命令是删除操作，该命令在复制重启时将被复制到 Slave 上，然后第二个语句是填充数据的语句。通过这种方法，可以确保即使 Slave 在内存表中含有过时数据，也不会导致 Master 和 Slave 的不一致。

Master崩溃导致二进制日志事件丢失

Master 可能发生错误，而没有将最近发生的事件写入到磁盘上的二进制日志中。也就是说，如果服务器在 MySQL 刷新其缓存到磁盘上（在二进制日志中）之前而崩溃，这些缓存事件将会丢失。

这通常表明 Slave 出错：二进制日志偏移事件丢失或不存在。在这种情况下，Slave 将在重启后使用已知的最后的二进制日志文件和 Master 的位置试图重新连接 Master，尽管二进制文件可能存在，但这些偏差并不是由增加的偏移事件没有被写入磁盘而导致的。

不幸的是，没有办法找回丢失的二进制日志事件。为了解决这个问题，必须检查 Master 上当前二进制日志的位置，并使用这个信息让 Slave 从 Master 的下一个已知的事件开始执行。为了确保 Slave 是同步的，请一定要同时检查 Master 和 Slave 上的数据。

还有可能是有些在 Master 上丢失的事件在系统崩溃之前被应用到数据上。你应该经常比较 Master 上有问题的表，以确定 Master 与 Slave 上的表之间是否有差别。这种情况很少，但是可能会导致后面的问题：如果 Master 上执行的一个行更新丢失了，这时在 Slave 上运行这个丢失的行更新将会导致错误。在这种情况下，Slave 将试图更新一个不存在的行。

390

例如，考虑一个虚构的场景：一个有关汽车经销商的简单数据库，该数据库包含新车和二手车销售信息的表，这些表被设置了自增键。

在 Master 上，发生了以下事件：

```
INSERT INTO auto.used_cars VALUES (2004, 'Porsche', 'Cayman', 23100, 'blue');
```

在执行下面语句之后发生了系统崩溃，但它还没有被写入二进制日志：

```
UPDATE auto.used_cars SET color = 'white' WHERE id = 17;
```

在这种情况下，该更新操作在 Master 崩溃时丢失了。当 Slave 尝试重启时错误发生了。可以使用以下建议解决该问题。查看表的行数在 Master 和 Slave 上是否相同。注意，更新操作将 2004 保时捷的颜色从蓝色换为白色。现在考虑当营业员试图帮助客户在 Slave 上查找蓝色保时捷时，会发生什么事情。

```
SELECT * FROM auto.used_cars
WHERE make = 'Porsche' AND model = 'Cayman' AND color = 'blue';
```

营业员将会找到蓝色的保时捷吗？出色的汽车销售员总是会用肉眼观察一下以确保现场有此车，但是我们假想一下，如果他太忙而没有时间去观察现场，而直接告诉顾客有他要的蓝色的保时捷。可以想像，当顾客在试驾车处发现车是白色时，将有多尴尬（失去了一个销售机会）。



为了防止数据在 Master 崩溃时丢失，请在系统启动时或在配置文件中启用 sync_binlog（将其设置为 1）。这将告诉 Master 需要立即刷新事件到二进制日志中。虽然这可能导致 InnoDB 性能下降，但是如果你不能承担失去任何数据（但是你可能丢失最新事件，这依赖于崩溃什么时候发生），这样做将是值得的。

虽然这个教学例子可能看起来并不严重，但是当医学数据库或科学数据库丢失更新事件时，甚至是个简单的更新都会给用户带来麻烦。事实上，以上实例可以被认为是数据损坏。当遇到问题时，请经常检查数据表的内容，在这种情况下，崩溃恢复可以确保当 sync_binlog=1 时二进制日志和 InnoDB 是一致的，但是其对 MyISAM 不起作用。

391

查询在Master上运行正常，在Slave上却运行不正常

有时查询（例如更新或插入命令）可能在 Master 上运行良好，而在 Slave 上不能正常运行，从严格意义上讲，这并不是 Master 的问题。这种类型的错误原因很多，但大多数是由完整性问题引起的，或者可能是 Slave 或数据库的配置出了问题。

这种错误的常见原因是数据库操作引用的表在 Slave 上不存在，或者表中的字段不同（不同的字段或字段类型不同）。在这种情况下，应该更改 Slave 使其与 Master 一致，以便能够正确地执行查询。

在有些情况下，查询语句引用了一个没有被复制的表。例如，如果你使用任何复制过滤启动选项（快速检查 Master 和 Slave 的确认该情况），有可能出现查询语句引用的数据库在 Slave 中不存在的情况。在这种情况下，应该调整相应的过滤器或者在 Slave 上手

动添加丢失的表到丢失的数据库中。

在其他情况下，失败查询的原因可能更复杂，如字符集问题、表损坏或数据损坏。如果你确定 Master 和 Slave 的配置一致，可能需要手动诊断查询。如果不能在 Slave 上纠正问题，可能需要手动执行更新操作并让 Slave 跳过包含失败查询的事件。



使用 `sql_slave_skip_counter` 变量在 Slave 上跳过一个事件，并指定想要跳过的来自于 Master 的事件。有时这是重新启动复制的最快办法。

系统崩溃后的表损坏

如果 Master 或 Slave 发生了崩溃，在重启它们后，你会发现一个或多个表被损坏，或者发现它们被 MyISAM 标记为 crashed，需要在重启复制之前解决这些问题。

可以通过检查服务器日志文件来发现哪些表被损坏，查找类似于下面的错误：

```
... [ERROR] /usr/bin/mysqld: Table 'db1.t1' is marked
as crashed and should be repaired ...
```

可以使用以下命令执行优化和一步修复给定数据库中的所有表（在本例中是 *db1*）。

```
mysqlcheck -u <user> -p --check --optimize --auto-repair db1
```



可以使用 `myisam-recover` 选项启动自动恢复 MyISAM 表。有四种恢复模式。详见 MySQL 在线参考文档以获得更多这方面的信息。

392

在修复了受影响的表后，还必须确认 Slave 上的表是否被损坏。如果 Master 和 Slave 共享同一个数据中心和配置环境（例如，它们被连接到同一个电源，这样做是必需的。



在修复前总是需要执行表备份。在某些情况下，修复操作可以导致数据丢失或将表留在一个未知状态。

另外，还有可能是修复导致 Master 与 Slave 不同步，尤其是修复导致数据丢失时。可能需要比较受影响的数据表中的数据，以确定 Master 与 Slave 是否同步。如果它们不同步，当 Slave 丢失数据时，你可能需要为 Slave 上受影响的表重新加载数据，或者当 Master 丢失数据时，将 Slave 的数据复制到 Master。

Master上的二进制日志被损坏

如果一台服务器崩溃或出现磁盘问题，将导致 Master 上的二进制日志被损坏，你将无法重启复制。二进制日志损坏的原因和类型有很多，但是任何损坏都会导致 Slave 上的一个或更多事件无法执行，经常导致不能解析中继日志事件。

在这种情况下，必须为恢复事件而仔细检查二进制日志，并使用 `FLUSH LOGS` 命令刷新 Master 上的日志。另外，出现以上情况时，Slave 可能会丢失数据，且大多数情况下必然会出现错误。最好的恢复办法是使用可靠的备份和恢复工具重新同步 Master 和 Slave。另外，刷新日志可以确保任何数据丢失最小化，并使复制重启时不出现错误。

在某些情况下，如果很容易就确认出有多少事件被损坏或丢失，可以使用 Slave 上的 `sql_slave_skip_counter` 选项跳过被损坏的事件。可以通过比较 Slave 上的 Master 的 binlog 和 Master 上的当前 binlog 位置来确认丢失或损坏的日志。

杀死长时间运行的非事务性表的查询

如果强制停止修改非事务性表，而该事件可能已经被复制到 Slave 上且被执行了，当这种事情发生时，Master 上的更新可能和 Slave 上不同。

393 例如，如果你中止以下查询：更新 600 行的表中的 400 行数据，此时 Master 中 400 行数据中只有 200 行被更新，这时 Slave 可能已更新完所有的 400 行数据。

因此，无论你什么时候中止 Master 上的数据更新查询，都需要确认该查询没有在 Slave 上执行，如果在 Slave 上已经被执行，一旦纠正了 Master 的表数据，就应该同步 Slave 上的数据。通常在这种情况下，将需要修正 Master，然后备份 Master 上的数据，并在 Slave 上恢复数据。

Slave上的问题

你遇到的大部分问题是由于 Slave 出现错误引起的。在某些情况下，如前面所述，有些问题来源于 Master，但是也会以这种或那种形式出现在 Slave 上。下面的章节描述了 Slave 上的一些常见问题。

使用 Slave 上的二进制日志

确保 Slave 更强大的一个办法是使用 `log-slaveupdates` 选项开启二进制日志。这将导致 Slave 记录中继日志中执行的事件，从而创建了一个二进制日志，在中继日志被损坏时，可以使用该二进制日志文件重放 Slave 上的事件。

Slave崩溃而且复制无法启动

当 Slave 崩溃时，如果确定了 Slave 上最后被执行的已知正确的事件，就会很容易与 Master 重新建立复制。可以在 SHOW SLAVE STATUS 命令的输出结果中查看该情况。

然而，如果错误发生在账号登录时，复制有可能不能被重启。这可能是认证问题导致的（例如 Slave 的复制账户被删除），也有可能是因为 Master 或 Slave 上的表被损坏了。在这种情况下，可以在 Slave 的 MySQL 服务器的控制台和日志上看到连接错误信息。

当发生这种情况时，总是需要检查 Master 上复制用户的权限。确保用户被授予正确的权限，该权限被定义在配置文件中或通过 CHANGE MASTER 命令定义，权限内容应该如下所示：

```
GRANT REPLICATION SLAVE ON *.*
TO 'rpl_user'@'%' IDENTIFIED BY 'password_here';
```

修改此命令以满足你的需要，是解决以上问题的一种手段。

Slave连接超时且重连接频繁

如果你的拓扑结构上拥有很多 Slave，且没有设置 server_id 选项或有两个以上的 Slave 含有相同的 server_id 值，将可能出现服务器 ID 冲突。当这种情况发生时，其中一个 Slave 可能会频繁超时或丢失后重新连接序列。

这个问题只是因为 Slave 没有唯一 ID，且该问题很难被诊断出来（或者，我们应该说它很容易被误诊为连接问题）。你应该总是检查 Master 的错误日志和 Slave 的错误信息。因为错误中有可能包含超时的真正原因。

为了防止这类问题的发生，总是需要确保所有服务器在配置文件中或在启动命令行中设置了 server_id。

相同的查询在Slave和Master上结果不同

一个更难以觉察的问题是：在一台或多台 Slave 上执行的查询结果与 Master 上的不一致。有可能你从来没有关注过此问题。该问题可能像排序问题一样简单或无害，也可能与在结果集中丢失额外行的问题一样严重。

该问题发生的主要原因是 Master 和 Slave 上的字符集设置不同。例如，Master 上的字符集为一种默认类型，而 Slave 的字符集被配置为其他类型。

如果用户为额外行、丢失行或不同的排序结果而抱怨，你应该首先检查 Master 和 Slave 上的字符集的设置。

该问题发生的另一个原因是在 Master 和 Slave 上使用了不同的默认存储引擎——例如，

Master 上使用 MyISAM 存储引擎，而 Slave 上使用 InnoDB 存储引擎。在这种情况下，如果你使用 ALTER TABLE 命令将 Slave 存储引擎换成与 Master 不同的类型，那么查询结果完全有可能以不同的顺序出现。

当 Master 和 Slave 上的表定义不同时，可能有更微妙的原因导致该类型的问题发生。有可能出现以下差异：一个给定表的字段子集是相同的，而有些初始字段或结束字段（顺序很重要）在 Slave 上丢失。

当你使用该功能时，将会出现很多潜在的错误，它有时会引起以下结果：有些字段的数据被复制而 Slave 没有定义该字段。当 Slave 上需要较少字段列时，粗心的用户可以通过删除字段的方法来达到以上效果，此时复制仍可继续进行。在某些情况下，当查询引用了丢失的列时，在 Slave 上执行 SELECT 查询将会失败，从而为你检查该问题提供了线索。有些时候，在应用程序中很容易丢失数据。

395 普通用户错误：在 Slave 上执行改变表或数据库的类型的操作，而在 Master 没有执行，这样会引起 Master 和 Slave 服务器上的查询结果不同，也就是说，用户在 Slave 上执行一些非复制的数据操作来更新表的特征，而在 Master 上没有执行同样的操作。当这种情况发生时，查询可能会返回错误的结果、错误的列、错误的顺序或额外数据，查询也有可能由于引用了丢失的字段而失败。检查包含在这种错误中的表的布局是一个很好的预防该问题的措施，这样可以确保 Master 和 Slave 的表一致。如果它们不一致，请重新同步 Master 和 Slave 上的表，然后重新执行查询操作。

Slave在尝试重启SSL时发生错误

有关 SSL 连接的问题是前面提到的典型的常见权限问题。在这种情况下，授权还必须包括 REQUIRE SSL 选项，如下所示。务必检查复制用户是否存在且拥有正确的权限。

```
GRANT REPLICATION SLAVE ON *.*
TO 'rpl_user'@'%' IDENTIFIED BY 'password_here' REQUIRE SSL;
```

当使用 SSL 连接时，其他有关重启复制的问题将出现以下情况：丢失认证文件，或配置文件中与 SSL 相关的选项（如 ssl-ca，ssl-cert 和 ssl-key）的值设置不正确，或 CHANGE MASTER 命令中与 SSL 相关的选项（如 MASTER_SSL_CA，MASTER_SSL_CAPATH，MASTER_SSL_CERT 和 MASTER_SSL_KEY）的值设置不正确。务必检查你的有关 SSL 的设置和路径，以确保自从最近一次启动复制后这些设置没有发生任何变化。

内存表数据丢失

如果你的数据库中的一个或多个使用了内存存储引擎，当 Slave 重启时，这些表里的数据将会丢失。这是意料之中的，因为内存表中的数据会在系统重启时被清除。这时，表

的配置仍然是存在的，且该表可以被访问，但是表里面的数据已经被清空了。

当 Slave 被重启时，直接针对于内存表的查询操作（如 UPDATE 操作）将会失败，或者查询结果不正确（如 SELECT 操作）。因此，错误可能不是立即产生的，而是可能在查询结果中出现行丢失。

为了避免该类问题，你应该小心使用数据库中的内存表。不应该这样做：在 Master 上创建内存表，并通过复制的方式在 Slave 上更新该表，而在服务器重启或事务崩溃后没有恢复该表的数据。例如，你可以在复制之前执行一个脚本，该脚本复制 Master 上的数据。如果这些数据被清除，可以使用脚本将数据重新填充到 Slave 的表中。

其他需要考虑的事情是：在复制过程中过滤表，或者对任何复制表不使用内存存储引擎。

临时表在Slave崩溃后丢失

如果你的复制数据库和查询使用了临时表，应该考虑一些有关临时表的重要因素。当 Slave 重启时，临时表将丢失。如果有任何临时表从 Master 被复制到 Slave 上，从那一刻起你不能重启 Slave，可能需要手动创建表或跳过引用了该临时表的查询操作。

这种情况经常导致查询没有在一个或多个 Slave 上执行。解决该问题的办法同解决内存表丢失的办法类似。特别是，为了使这些查询被执行，你可能需要手动重建临时表，或者将 Slave 上的数据与 Master 上的数据重新同步，并在重启 Slave 的时候跳过这些查询。

Slave运行过慢且其不能与Master同步

Slave 滞后，也被称为 *excessive lag*，是指 Slave 不能快速执行来自于 Master 的所有事件，从而不能避免更新数据的延时。在极端情况下，Slave 上的数据更新因为超时过期而产生不正确的结果。例如，如果票务代理系统中的 Slave 比 Master 延迟几分钟，该票务系统可能会出售不存在的座位（例如，这些座位在 Master 上已经被标记为“已卖”，而 Slave 由于延时而未被更新）。

我们在前面的章节讨论过该问题，但是在这里仍然简单概括一下解决方案。为了检测该问题，需要监控 Slave 的 SHOW SLAVE STATUS 输出结果，并需要检查 Seconds_Behind_Master 字段以确保 Seconds_Behind_Master 值在应用程序允许的范围之内。为了解决该问题，需要考虑将一些数据库移植到其他 Slave 上，从而减少复制到 Slave 的数据库数量，并可以减少网络延迟，提高数据存储性能。

例如，可以为大数据集的更新提供额外的 Slave，从而使当前 Slave 可以处理其他事务。还可以通过以下方法减轻复制负担：在一个单独的 Slave 上执行数据更新，在拓扑结构上的所有其他机器上使用可靠的备份和恢复方法来执行更新。

Slave崩溃后数据丢失

Slave有可能崩溃而没有记录最近可知的Master的二进制日志的位置。这些信息存储在*relay_log.info*文件中。当这种情况发生时，Slave将在错误（老）位置试图重新启动，且试图执行一些已经执行过的查询。这常常会造成查询错误，可以通过跳过重复事件来处理该问题。

然而，这些重复事件也可能会导致数据被改变（被损坏），致使Slave与Master不同步。不幸的是，这类问题不太容易被侦查出来。留心检查日志文件可能发现有些事件已经被
397 执行过了，可能需要检查Slave的binlog事件和Master的二进制日志以确定哪些事件是重复的。

Master崩溃导致表损坏

当在崩溃后重启Master后，你可能会发现其中有些表已经被损坏或被MyISAM标记为被损坏。必须在重新启动复制之前解决这些问题。一旦修复了被影响的表，就需要确保Slave上该表没有因为修复而出现数据丢失。这种情况很少发生，但是还是应该确认一下。当出现疑问时，常常需要手动将这些表与Master同步，在重启复制之前使用备份和恢复或相似操作实现数据同步。



在硬件或服务器崩溃的过程中，当部分页面发生写操作时，MyISAM上执行恢复操作之后可能出现数据丢失。不幸的是，数据是否丢失不太容易被检测出来。

Slave上的中继日志被损坏

如果服务器崩溃或出现磁盘问题，将会导致Slave上的中继日志损坏，复制将会因为几个与中继日志相关的错误而停止。中继日志损坏的原因和类型很多，但是都会导致Slave上的事务不能被执行。

当这种情况发生时，恢复的最佳选择是：需要确认来自于Master的二进制日志中的最后一个被执行的事务，并需要使用`CHANGE MASTER`命令重启复制，且需要提供Master的binlog信息。这将强迫Slave重新创建新的中继日志。不幸的是，这将意味着旧中继日志的任何恢复可以被妥协。

Slave重启时出现大量错误

难以察觉和修复的困难问题之一是：在Slave初始启动或重新启动时产生大量错误。这一系列的错误有时候是随机发生的，或者没有明确的可识别原因。

当这种情况发生时，请检查 Master 和 Slave 上的 max_allowed_packet 的大小。如果 Master 上该参数的值比 Slave 上的大，有可能因为 Master 上记录的事件的大小超过了 Slave 上的大小。这将会导致看起来不合逻辑的随机错误发生。

Slave上的失败事务导致的结果

一般情况下，当有失败事务发生时，更新被回滚以避免部分更新问题的发生。然而，该问题相当复杂，当你混合了事务性和非事务性表时，事务性更新被回滚，但是非事务性更新却没有被回滚。这样可能导致以下问题发生：数据丢失、数据重复、数据冗余或非事务性表的无用更新。

398

避免该类问题的最好办法是避免混合数据库中事务性和非事务性表的关系，并且需要经常使用事务性存储引擎。

高级复制问题

在执行一些更高级的复制拓扑结构时，会出现一些并发的问题。本节将讨论一些在使用高级复制功能时遇到的问题。

更新没有在拓扑结构中被复制

在某些情况下，某个数据库的改变没有被复制。例如，ALTER TABLE 可能被复制，但是 FLUSH、REPAIR TABLE 及类似命令没有被复制。不论该情况何时发生，请查阅数据控制命令（DML）和维护命令的限制因素。

导致该问题发生的原因是：无经验的管理员或开发者试图管理 Master 上的数据库，并希望 Master 上的更新被复制到 Slave 上。

不论数据库对象何时发生巨大的改变，如在文件级别改变其结构或使用维护命令改变数据库，请在所有的 Slave 上执行这些命令或程序，以确保 Master 的数据库更新在这个拓扑结构中传播。

专业的管理人员经常使用脚本去完成日常维护类的工作。通常情况下，脚本用于按顺序停止复制、应用更新改变和自动重启复制。

循环复制问题

如果你使用循环复制，并已经恢复了复制错误，且该复制错误是由于一个或更多的服务器从拓扑系统中取出而引起的，此时，会遇到这样的问题：某个事件在有些服务器执行多次。如果查询出错（如键冲突），将导致复制失败。这种事情发生是因为源服务器也

被移出拓扑系统。

当这种情况发生时，源服务器没有终止复制事件。可以使用 IGNORE_SERVER_IDS 选项（MySQL 5.5.2 中可使用）和 CHANGE MASTER 命令解决该问题，该方法可以提供忽略事件的服务器 IDS 列表。当丢失的服务器被重启时，必须调整该设置，使来自于代理服务器的事件不会被忽略。

399

多Master问题

当使用循环复制（多 Master 拓扑结构的特殊形式）时，如果你正在恢复复制错误，可能会发现有些事件被执行多次。这些事件一般情况下来自于被移出的服务器。可以使用解决循环复制的方法来解决该问题——使用 CHANGE MASTER 命令将被移出的服务器的 IDS 存放在 IGNORE_SERVER_IDS 选项列表中。

多 Master 复制的另外一个问题一般发生在以下情况下：当对同一个表的改变同时发生在所有 Master 上，且该表的主键为自增列时。在这种情况下，你会遇到重复主键错误。如果必须在多个 Master 上插入新行，请使用 auto_increment_increment 和 auto_increment_offset 选项进行主键值的递增。例如，某个服务器仅仅只能以偶数递增，而其他服务器却以奇数递增。当使用该办法解决了多 Master 的问题后，在多个 Master 上使用自增主键更新同一个表将变得比较复杂。不仅使键值自增变得困难，而且会在你需要替换拓扑结构中更新该表的服务器时出现管理问题。例如，可以在自增值中结束差距，这将最终导致表中的键值大于该列数据类型的最大值。

HA_ERR_KEY_NOT_FOUND错误

该错误在基于行复制的拓扑系统中很常见。最有可能导致该错误发生的原因是：需要被更新或删除的行并不存在或已经被改变，以至于存储引擎找不到它。该错误主要发生在循环复制中，或发生在直接改变 Slave 上的复制数据时。当这种情况发生时，必须找出冲突的根源并修复数据或跳过引起问题的事件。

排除复制故障的工具

如果你已经使用或建立了复制或性能维护，有很多用于诊断和修复复制问题的工具是你所熟悉的。

本节将讨论诊断复制问题的工具，并提供一些如何及何时使用这些工具的建议：

- SHOW MASTER STATUS 和 SHOW SLAVE STATUS
 这些 SQL 命令是诊断复制问题的主要工具。它们与 SHOW PROCESSLIST 命令一起

使用，应该先在 Master 上执行这些命令，然后在 Slave 上执行，再检查输出结果。Slave 命令中的扩充参数集在诊断复制问题时没有多大用处。

- SHOW GRANTS FOR（复制用户）

 不论何时你遇到 Slave 用户访问问题，应该首先检查该 Slave 用户的权限以确认它们没有被更改过。

400

- CHANGE MASTER

 有时候配置文件在知情或不知情的情况下被改变了。请使用 SQL 命令覆盖撤销最后可知的连接参数，并诊断 Slave 连接问题。

- STOP/START SLAVE

 使用这些 SQL 命令启动和停止复制。如果 Slave 出现错误，有时候最好的办法是中止运行该 Slave。

- 检查配置文件

 有时候问题的发生是由于未批准或遗忘的配置更改而造成的。请在诊断连接问题时例行检查配置文件。

- 检查服务器日志

 应该在诊断问题时习惯性地检查服务器日志。检查服务器日志有时可以发现一些不可见的错误。这些错误和警告信息是非常有用的。

- SHOW SLAVE HOSTS

 使用该命令识别 Master 上连接的 Slave，如果它们使用 report-host 选项。

- SHOW PROCESSLIST

 当遇到问题时，最好去查看一下其他运行的进程。该命令将告诉你有关复制的每个进程的当前状态。当遇到问题时，请先查看复制进程信息。

- SHOW BINLOG EVENTS

 该 SQL 命令显示二进制日志中的事件。如果你使用基于语句的复制，该命令将显示使用 SQL 语句的更改信息。

- mysqlbinlog

 该工具允许读取二进制日志或中继日志中的事件，它常常显示什么时候出现了损坏事件。在诊断与事件和二进制日志相关的问题时，请毫不犹豫地使用该工具。

- PURGE BINARY LOGS

 该 SQL 命令允许移除二进制日志中的某些事件，如移除在特定时间后或给定事件后发生的事件。你的例行维护计划应该包括使用该命令清除不再使用的旧二进制日志。

目前我们已经回顾了在复制中可能遇到的一些问题，并参看了在常用 MySQL 版本中可获得的诊断工具列表，现在将讨论解决复制问题的策略。

401

最佳实践

回顾可能在复制中发生的潜在问题，并列举解决这些问题的工具，这些只是完全解决复制问题的一部分。这里有一些经过验证的策略和最佳实践可用于快速解决复制问题。

本节描述了一些用于诊断和修复复制问题的策略和有助于诊断和修复复制问题的最佳实践。我们不是按特殊顺序来讲解这些方法的——取决于你试图解决的问题，其中有一个或多个可能会对你有所帮助。

了解你的拓扑结构

如果你在少数服务器上使用 MySQL 复制，将拓扑配置存储在内存中并不是什么难事。可能与单 Master 和一个或多个 Slave 的拓扑结构一样简单，或者也有可能与有两个服务器的多 Master 拓扑结构一样复杂。然而，要记住拓扑结构和所有的配置参数都变得不可能。

拓扑结构和配置越复杂，就越难确认问题发生的原因及从何开始修复操作。在一个有上百台 Slave 的拓扑系统中很容易忽略单个 Slave。

最好是拥有一份拓扑结构的图谱及其配置设置。应该在记事本或文件中记录一份复制启动的相关信息，并将该记录放在大家比较容易找到的地方。

你应该拥有一份拓扑系统的文本或图谱，并在上标明过滤器（Master 和 Slave），还应该标明各个服务器在拓扑结构中的角色。还应该考虑包含 `CHANGE MASTER` 命令、所有的选项及所有服务器的配置内容的文件。

拓扑结构的图谱不需要很详细或是一幅艺术奇作。简单的线条描画就比较好了。图 11-1 显示了一幅混合拓扑结构图，且图上布满了过滤器和角色记号。

请注意中继 Slave（192.168.1.105）有两个 Master（192.168.1.100 和 192.168.1.101）。这显得很奇怪，因为没有一个 Slave 可以拥有一个以上的 Master。为了达到这种级别的整合（消耗第三方数据），你将在中继 Slave 上需要第二个 MySQL 服务器的实例，并使用该实例复制来自于战略伙伴（192.168.1.101）的数据，还需要在中继 Slave 上使用脚本定期将 MySQL 的第二实例上的数据转移到 MySQL 的主实例上。这样将可以达到图 11-1 中描述的集成方式，包含一些手动工作和战略伙伴数据的时间延迟性更新。

402 特定的拓扑系统会产生固有的问题。本章和前面的章节已经讨论了大多数这些问题。下面将介绍拓扑结构的类型，其中包括这些类型的简单描述和系统的一些漏洞。

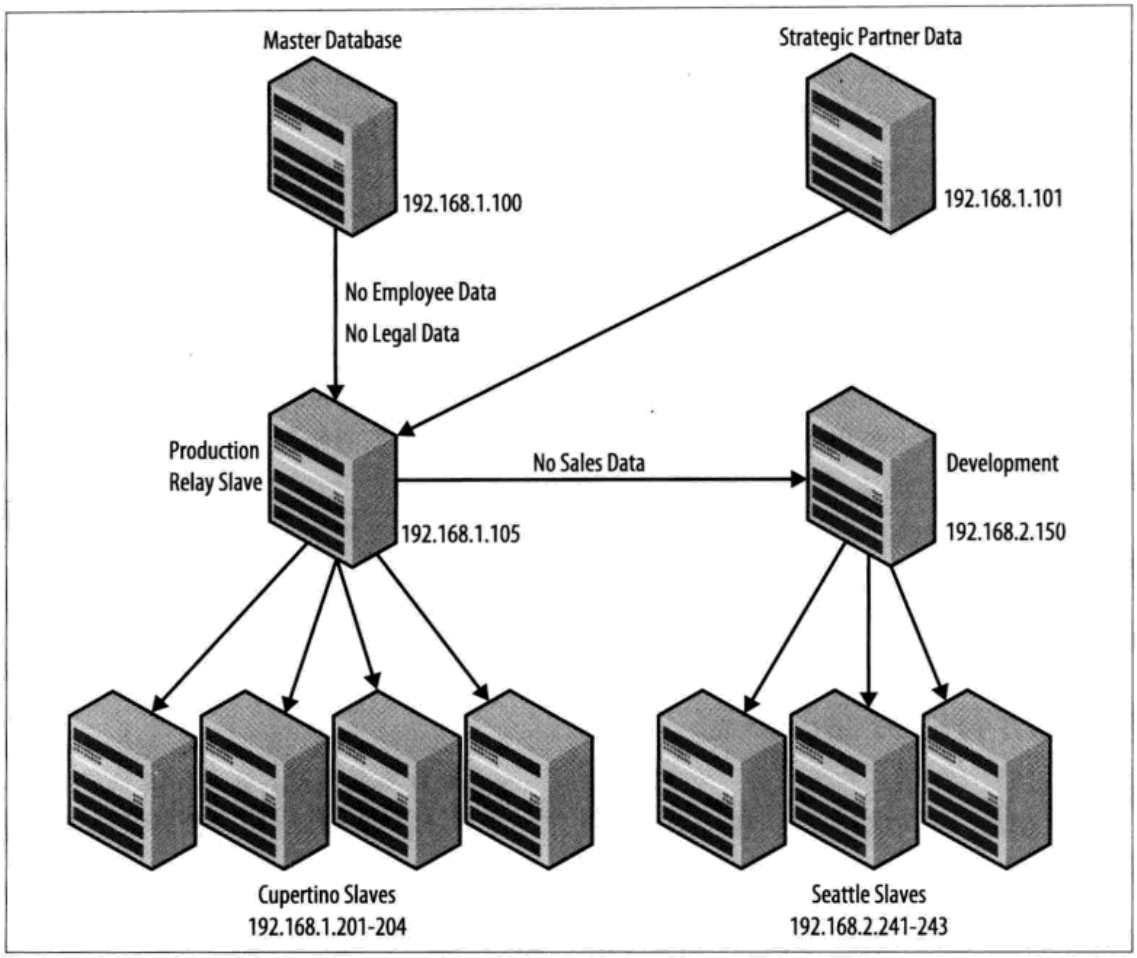


图11-1： 拓扑图例

- 星形（也被称为单个 Master）
 这是典型的单 Master 和多 Slave 的拓扑结构。除适用于作为一个整体复制外，没有什么特殊的限制和问题。然而，将该结构中的一个 Slave 提升为 Master 将会比较复杂，必须将最新的 Slave 提升为 Master。这将很难决定且可能需要检查每个 Slave 的状态。
- 链式
 在这种结构中，一个 Slave 是另外一个 Slave 的 Master，如此向后推移，最后一个端点只是简单的 Slave。除开前面提到的服务器 ID 问题，当链中间的主 Slave 发生错误时，确定将其放在何位置也是个问题。提升一个新的 Master/Slave 节点将需要额外的工作以确保整个系统的一致性。 403
- 圆形（也被称为环形）
 该结构与链形类似，但是没有结束点。该拓扑结构需要用心建立以使事件能在启始服务器上终止。

- 多 Master
 它与环形拓扑结构类似，但是多 Master 上的每个 Master 是其他每个 Master 的 Slave。该拓扑结构含有环形复制的所有限制和问题，且由于会出现冲突更改而成为难以管理的复杂拓扑结构之一。为了避免该冲突，必须确保仅仅更改 Master 中的一个。如果发生冲突更改，请在一个 Master 上执行 UPDATE 命令，在另一个上执行 DELETE 命令，DELETE 命令有可能在 UPDATE 命令之前执行，这将会导致 UPDATE 命令失败。
- 混合型
 混合型拓扑结构使用其他拓扑结构中的一些或所有元素。经常会在大型组织中看到该结构，该大型组织的各分隔功能需要基础结构中类似的但常常独立的元素。隔离是通过过滤完成的，这时数据从原始 Master（有时被称为 prime 或 Master 的 Master）中被划分到多个 Slave 中，该原 Master 包含了发送到 Slave 上的所有数据，这些 Slave 有成为自身星形拓扑的 Master。它是目前最复杂的拓扑结构，需要常常认真更新文档，以获取最新信息。

查询所有服务器的状态

预防问题发生的最好的措施之一是定期检查拓扑结构中所有服务器的状态。该工作不会很复杂。例如，你可以建立一个调度任务，该任务可以启动 MySQL 客户端，并发出 SHOW MASTER STATUS 或 SHOW SLAVE STATUS 命令，然后输出或以邮件形式发送出结果。

当然，你需要阅读这些电子邮件信息！建议建立调度任务并在你有时间阅读信息之前运行它。例如，你可以在午饭之前运行该任务，然后在午饭时间或午饭后阅读任务输出的结果。

监控服务器状态是获取潜在问题的很好的办法。它还使你有机会快速应对错误的发生。

你应该查找错误、监控滞后的 Slave、检查过滤器以查看它们是否与记录相符，并确保所有的 Slave 在运行且没有报告警告信息或连接错误。

404

查看日志

除定期检查服务器状态外，还应该考虑定期检查服务器的日志。可以使用以下方法方便地查看日志信息：使用 MySQL 管理器图形用户接口连接每个服务器并查看日志中的最新条目。

坚持不懈地执行此工作，将有利于前期的问题侦查。有时候某些错误或警告没有在任何侦测中体现出来却被记入了日志中。早点发现这些问题信号可以使修复工作更容易。

建议在查询服务器状态的同时检查日志信息。

检查配置信息

除开检查服务器状态和日志文件外，你还应该例行检查配置文件信息，以确保文档信息是最新的。该项检查没有日志检查那么重要，而且往往容易被忽视。推荐检查配置文件，另外，如果对拓扑结构或服务器做了很多更改，建议每周至少更新一次文档，如果对拓扑结构或服务器做了很少的更改，建议每月至少更新一次文档。如果拓扑结构环境中不止一个管理员，可能需要考虑频繁做上面的工作。

有序执行系统关闭

有时候，在诊断和修复问题时关闭复制是必要的。如果复制已经被关闭，可能不需要做任何事情，但是如果你有一个复制的拓扑系统，且错误与数据丢失相关，这时通过拓扑结构停止复制可能比较安全。但是应该在可控的和安全的情况下做此事。

有很多策略用于执行复制拓扑的可控关闭。如果数据丢失而 Slave 又没有延迟，可能需要先锁住 Master 上的表并刷新二进制日志，然后等待所有的（剩余的）Slave 跟上，再关闭 Slave。这将确保所有事务在可使用的 Slave 上被复制和执行。

另外，如果问题比较严重，可能需要启动拓扑结构的叶分支上的 Slave，并在拓扑结构上执行你的工作，保持 Master 在运行中。然而，如果保持 Master 在运行状态中（例如，你没有锁定所有的表），且更新操作的负载很重，以及系统的诊断和修复花费很长时间，那么在重启复制时，Slave 将滞后于 Master。在这种情况下，如果你认为修复将花费很长时间，那么最好停止 Master 上的更新。

如果你面临一个很困难的问题，或者更糟糕的是，该问题是在没有任何征兆的情况下随机发生的，你可能需要考虑完全关闭复制。这种情况往往发生在只有一个 Slave 遇到问题的时候。关闭复制将允许你在诊断问题时隔离该出问题的服务器。 405

在遇到错误后按序执行重启

按序重启复制也是很重要的。最好在可控的情况下重启复制拓扑，如在单 Master 和单 Slave 的拓扑结构上。即使你的拓扑结构更复杂，优先启动最基础的复制块将允许你做测试，并确保问题在所有服务器重启之前被修复。

当处理一些限制在某个服务器或一系列事件上的问题时，隔离操作是必要的。这也有利于启动单 Master 和单 Slave 的结构。在这种情况下，如果你不能够修复问题，将更容易从 MySQL 专家那获得帮助，因为你已经将问题缩小到尽可能小的参数集上。查看后面

的“报告复制错误”可以获取更多的细节信息。

手动执行失败查询

比较容易忽略的一件事是为了找出错误发生的线索而检查中继日志或二进制日志中的查询。很容易查找出发生在 Slave 上的错误，并诊断出可能出错的事情，但是有时候（尤其当包含了查询操作时）你可以通过隔离有问题的 Slave 和试图手动执行查询操作来得到更多与错误相关的信息。

如果使用基于语句的复制，这将是件很容易做的任务，因为该查询操作在二进制日志或中继日志中是可读的。如果你使用基于行的复制，仍可以执行该查询操作，但是不能读懂该查询本身。在这种情况下，一个畸形的或使用了丢失的、不正确的或损坏引用的查询操作可能不是那么易懂，直到你手动执行了该语句才明白其意思。

请记住，应该总是在试图运行可能会导致数据更改的诊断之前备份数据。运行受损的查询将会导致数据更改。我们已经看到有些导致复制错误的查询操作有时候手动可以执行成功。当这种情况发生时，常常表明 Slave 的配置出错了，或二进制日志或中继日志出错了，而不是查询操作本身出错了。

406

常用程序

还有一些常用的程序可以用于执行已经讨论过的实践，包括 Master 上的复制的故障排除和暂停复制。

复制失败的故障排除

排除复制问题的步骤如下所示。其中大部分步骤已经在本书的其他部分介绍过，或者你可能已经对它们很熟悉了。



你应该在执行这个流程时，记录下所有的观察数据。统一在一个地方查看这些信息有时候可以更清晰地看到问题发生的原因的全貌。

1. 检查 Master 的状态，并记录其异常信息。
2. 检查 Slave 的状态，并记录其异常信息。
3. 阅读所有服务器上的错误日志。
4. 检查每个服务器上的进程列表，查找状态中的异常信息。

5. 如果没有错误报告，请试图重启 Slave，并记录出现的任何错误。

6. 检查有问题的服务器的配置信息，以确保没有什么被改变。

7. 检查笔记，并为检查和修复问题制定计划。

该流程虽然过于简单化，但是它可以帮助你快速诊断复制问题。

暂停复制

为了暂停复制，请在每个 Master 上执行以下步骤，从第一个 Master 开始。

在 Master 上：

1. 运行 `FLUSH TABLES WITH READ LOCK` 命令。

2. 运行 `SHOW MASTER STATUS` 命令。

3. 记录 Master 的二进制日志和位置。

在 Slave 上：

4. 运行 `SELECT MASTER_POS_WAIT(binary_log_name, position)`。

在每个连接到 Master 的仍保持活动的 Slave 上，查看 `SHOW MASTER STATUS` 命令的结果直到 Slave 与 Master 同步。一旦 Slave 达到同步的状态，这时关闭 Slave 是安全的。

当你重启复制拓扑时，所有的 Slave 将自动启动而没有延迟。该流程在以下情况下是有用的：当系统发生损坏时，你希望使事情再次执行以避免 Slave 延迟。 407

报告复制故障

有时候你可能遇到一个在没有外界帮助的情况下无法解决的复制问题。即使你没有适当的支持协议，但是你可以使用 MySQL 进行故障报告。

最好的故障报告描述了可以使用独立的测试实例演示或重复的错误，例如，一个复制问题可以在测试的 Master 和 Slave 上被演示，且该问题的演示只需要少量数据集。这往往只不过是关于有问题的配置和事件的完整且精确的报告。

为了报告故障，请访问 *http://bugs.mysql.com*，然后（如果你没有登录过）请在该网站注册一个账户。当上传了故障报告时，请一定要完整地描述该问题。为了快速得到结果和解决办法，请在问题描述中尽量包含以下细节信息：

- 一字不差地记录所有的错误信息。
- 包含二进制日志（全部或摘录）。
- 包含中继日志（全部或摘录）。
- 记录 Master 和 Slave 的配置。
- 复制 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令的结果信息。
- 提供所有错误日志的副本。



请务必使用上面介绍的所有技术并使用所有可获得的工具探索问题。

不论故障报告何时被更新，MySQL 故障跟踪工具都将以邮件警告的方式发送故障报告。

小结

本章介绍了在使用复制时遇到的常见问题的解决办法，探究了复制失败的原因并讨论了如何解决这些问题以及在将来如何预防这些问题。还介绍了 MySQL 复制的故障排除的工具和最佳实践。

408

当听到老板正向他的门走过来时，Joel 笑了，此刻他已经准备好了怎样向老板报告。刚一看到 Summerson 先生时，Joel 就开始说："我已经在线恢复了西雅图的服务器，先生。这是由于 Slave 上表已经发生了改变，从而导致某些查询失败而引起的问题。我已建议西雅图员工将所有的 schema 更改和表维护直接发送到这儿的 Master 上，我将从现在开始进行观察，然后让您知道该问题是否还会发生。"

Summerson 先生笑了笑。Joel 感觉到前额都流出了汗珠。

"好，干得好。"

Joel 如释重负："谢谢你，先生。"

老板正准备离开，却又停了下来，转过身，说："还有件事情。"

"先生？" Joel 快速说道。

"是 Bob"。

Joel 有些疑惑。Summerson 先生的任务常常包含一个完整的句子。

“先生？”他犹豫地说。

“你可以叫我 Bob。我认为你最好去掉‘先生’这个称呼。”

Joel 静坐了一会儿，直到他意识到自己看着空白处发呆半天，他眨了眨眼，看到开发部门的好友 Amy 站在了他的办公桌前。

“你还好吗，Joel？”

“什么？”

“你在发呆，你在想什么呢？”

Joel 笑了起来并翘起了坐椅。

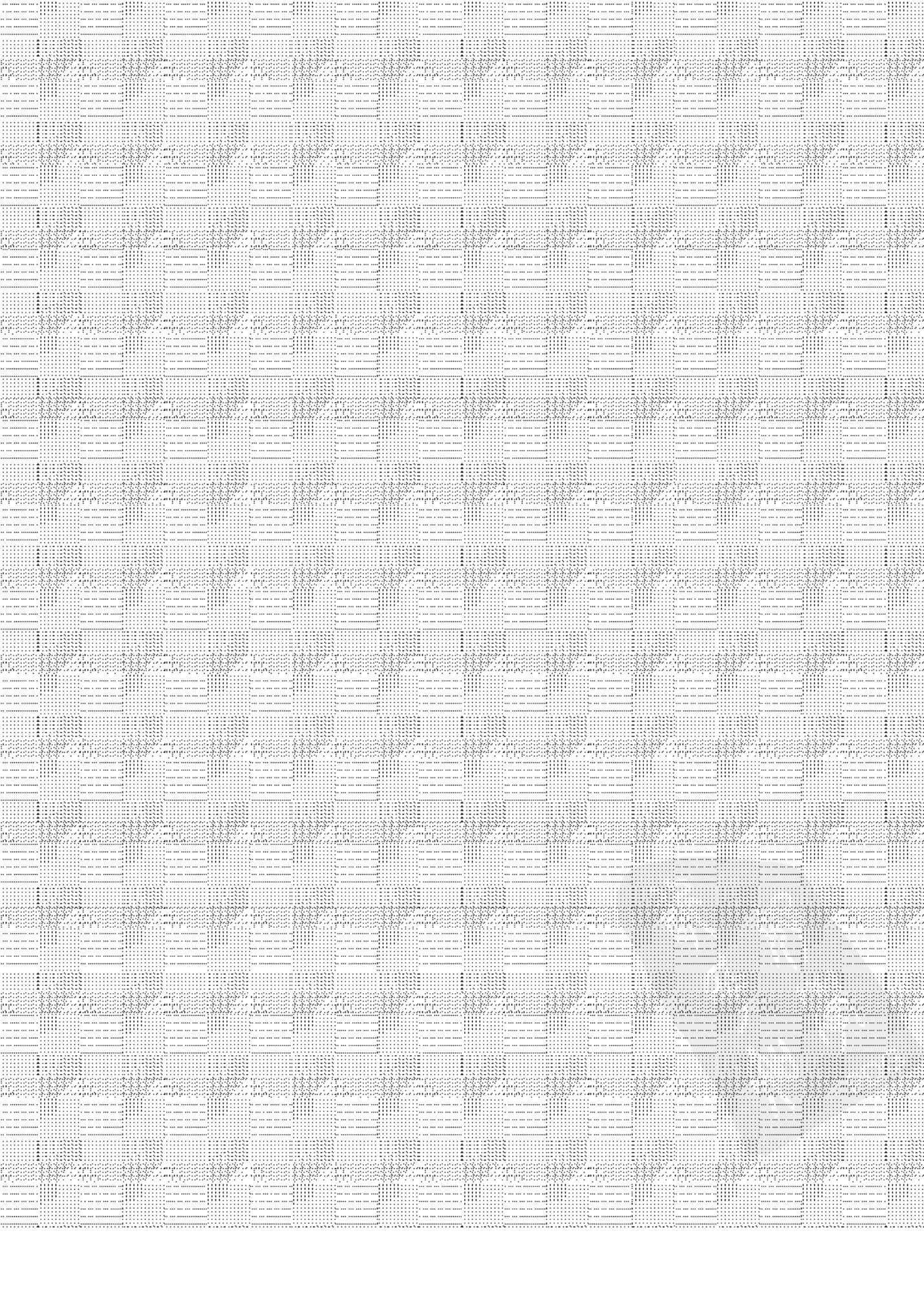
“我认为我最终征服他了。”

“谁？ Summerson 先生？”

“是的，他刚才让我喊他 Bob。”

Amy 笑着挽起自己的胳膊，然后开玩笑地说：“好的，我猜想以后要喊你 Joel 先生。”

Joel 笑着说：“吃过中饭了吗？”



第12章 409

保护你的投资

Joel 打开桌子中间的抽屉找钢笔，这时他从门缝中听到一个熟悉的声音："Joel，我需要你与信息保障审核员（information assurance auditors）谈谈，告诉他们我们的恢复计划。还有，再订些磁带，好让他们知道我们已经做了准备。"

"审核员？磁带？"与 Summerson 先生目光相触时 Joel 问道。

"务必开始致力于他们要求的任何灾难性恢复计划文档。还有，告诉采购人员你的文档需要什么类型的媒体。他们会告诉你填什么表格。再额外预备一些以防万一。"Joel 的上司说道，接着就离开了。

Joel 关上抽屉后站起来，心想老板指的是备用媒体。但是审核员和计划又是什么意思呢？Joel 将椅子拉到电脑前坐下。"嗯，我想我必须搞清楚什么是信息保障，以及如何做备份。"他边打开浏览器边喃喃自语。

Summerson 先生把头伸到门口说道："对了，Joel，希望明天早上你能把恢复计划放到我的桌子上，在 14:00 的连续性业务规划会议之前。"

Joel 叹了一口气，他意识到老板需要的远远超过他的想象。Joel 伸手拿起了 MySQL 书本。"这钱花得值。"然后翻开书中"保护你的投资"一章。

本章主要介绍数据保护和数据恢复，主要讨论备份计划、数据恢复和 MySQL 中复制和修复的流程。这些话题本章都没有深入讨论，没有对信息保障、信息的完整性、灾难恢复及相关概念进行介绍。事实上，一旦阅读了整个章节，你将学习到很多关于保护你的投资的知识，而不是只知道备份和恢复数据。

410

什么是信息保障

本节介绍信息保障的概念、对组织的意义，以及如何进行良好的信息保障。这里不打算全面介绍所有内容，但是会介绍一些基本知识，使你可以为学习该课题做好预备。对了，你还可以向老板介绍它呢。

信息保障（IA）是对信息系统技术的管理，旨在管理、监控、确保信息系统的可用性及安全控制等。IA 的目的是控制和保护数据和系统、访问权限，以及信息的机密性和可获得性。虽然本章主要针对于数据库系统，但是 IA 适用于信息技术的所有方面。以下网站包含了有关 IA 的使用背景知识：

- *http://www.nsa.gov/ia/*
- *http://en.wikipedia.org/wiki/Information_assurance*

信息保障的三个应用

有关研究人员将 IA 应用分为三类：

- 信息安全
 通过管理内部和外部的风险威胁来保护计算机系统。
- 信息完整性
 保证系统及其数据的持续可用性和连续性（有时被称为数据保护）。
- 信息价值
 用户对系统的价值评估。

其中信息安全最常见且记录最完善。如今有无数的参考资料是关于如何最好地维护计算机系统的安全的，还有许多参考资料是关于物理安全措施方面的。

然而，信息完整性是这三者中最重要、但也最容易被忽略的。本章主要介绍信息完整性的最佳实践的重要性。

信息的价值通常是被了解最少的应用。具有全面质量管理及相关学科的组织比较熟悉系统价值评估，但是一般只考虑企业的财政方面，而没有考虑与员工工作有关的信息的价值。尽管有人说对公司有利就是对员工有利，但我们也遇到过这样的情形：有些看起来似乎用心良苦的政策，却可能阻挠了员工的工作。

411 在考虑信息完整性及其价值时，IA 还涉及灾难计划和灾难恢复，而它们往往被忽略。

良好的 IA 计划包括非物理的（数据、访问码等）和物理的（计算机、访问卡、钥匙等）两方面。本章专注于物理方面，因为数据库系统可能是组织最重要的资源之一。

信息保障为什么重要

随着组织计算机系统的重要性的提升及技术成本的提高，节约花费和未雨绸缪成了优先考虑的因素。因此，IA 在研究和政府机构（最有兴趣的两个机构）之外的领域的应用也得到了推广。相关的概念越来越受欢迎，在很多公司的审核员的工具箱中找到了用武之地。

作为信息技术的领导者，你需要为企业的不可预测的危难（其中包括物理的、非物理的、财政威胁）做准备，保护企业免受灾难。一个常见的选择是使用 IA。如果你的组织已经或正在计划启动自己的 IA 计划，应该学习一下 IA 将如何影响你。

信息完整性、灾难恢复及备份的职责

信息完整性有时又称业务连续性，保证组织在不中断的情况下完成任务，并且可以对一个意想不到的灾难性事件进行可控恢复。这种灾难性事件可以是一个小问题，也可能是操作或数据的毁灭性丢失，后者无法快速解决（如断电恢复）或者缺少资金（如更换信息技术系统）。重要的是需要认识到没有单一的方法可以帮助你预防或恢复所有事件。



记住，硬件可能失效，最终也一定会失效，所以请提前做好准备。

信息完整性包括以下几个方面：

- 数据完整性
 确保数据永远不丢失，且始终保持最新版本，不被损坏。
- 通信完整性
 确保通信链路始终可用，且在中断的情况下可被恢复，可以启用一个可信连接。
- 系统完整性
 确保可以在系统故障的情况下重启系统，在丢失或损坏的情况下可以恢复数据和服务器状态，保持系统的连续性。

412

随着业务越来越依赖于其信息系统，系统在业务运营中的作用也变得越来越重要。事实上，大多数使用信息技术的现代业务已经变得如此依赖这些系统，使得业务及其信息技术已合为一体，也就是说，业务无法脱离信息系统而存在。

如果这变成事实，组织必须认识到如果其信息系统无效，它的业务将不能有效地运作（或者可能完全无法运作）。在这种情况下企业需要尽快恢复系统。在发生突发事件时恢复操作能力及数据，称为灾难恢复（*disaster recovery*）。

在监管机构审查下的企业知道（或者即将知道），法律决策和标准可能要求公司采用 IA 和灾难恢复措施。更重要的是，保护数据的工作已经取得一定进展。在美国，这些工作包括 1996 年的健康保险携带和责任法案（HIPAA）、爱国者法案和 2002 年的萨班斯 - 奥克斯利法案（SOX）。



请参看 *http://www.soxlaw.com/* 以获取更多的有关萨班斯 - 奥克斯利法案的信息。

高可用性与灾难恢复

大多数企业认识到他们必须进行技术投资，使系统能够从轻微或中度事件中迅速恢复。这些技术包括复制、廉价磁盘冗余阵列（RAID）、冗余电源供应等。这些都是高可用性选项，因为它们允许在没有任何数据丢失（或者尽量将数据损失减小到最小）的情况下实时或近实时地恢复系统。

不幸的是，很少有企业采取额外的措施保护他们的投资免受毁灭性的（且昂贵的）损失。那些采取措施的企业使用风险分析的最终形式，即灾难恢复来保护他们的投资。

413 当然，在高可用性和灾难恢复之间存在重叠。高可用性解决办法可以解决小灾难，甚至可以形成防御主要灾难的防御层。但是两者提供的保护类型不同，高可用性抵御已知或可预知的事件，而灾难恢复可以解决意料之外的灾难。

灾难恢复

灾难恢复涉及过程、政策及灾难性事件后的信息完整性的规划。最重要的是创建和维护灾难恢复计划文档。

这里将介绍灾难恢复的各个方面，本节给出了这些方面的概述，这样你可以向你的管理团队说明灾难恢复的重要性。

第 2 章已经讨论了一种灾难恢复形式。这些有时候是组织机构用于恢复小型灾难的前期工具，但是如果情况真的很糟糕怎么办，比如 RAID 阵列出现了无法恢复的故障，或者服务器被烧毁？

在做好最坏的打算之前，你需要回答一系列问题，这些问题构成了灾难恢复计划的前提或目标，同时也是评价计划有效性的标准。

- 你的组织机构面临哪些物理的和非物理的威胁？

- 你的业务需要什么样的操作能力进行维持？
- 你的业务在系统恢复之前能持续多久？
- 哪些资源可以用于计划和执行灾难恢复？

灾难恢复的第一步是认识到最坏情形：由于灾难性事件而导致的整个数据中心损失。不仅包括技术损失（服务器、工作站、网络设备等），还包括计算设施本身的损失，还有不要忘记操作系统的人员损失（这一点很少被考虑到，但是应该被考虑）。

虽然这看起来像是世界末日，但是现实中已经发生过并可能重复发生的事件可能会带来类似的破坏。它可以像大范围停电一样简单，也可能像飓风、地震甚至战争一样具有灾难性。

想象一下，信息技术所在的大厦由于严重的暴风雨而被损坏且无法修复。屋顶被撕裂，90% 的物理资源被水和掉落的碎片损坏。再假设你的公司正在进行一项赚钱的业务交易，并且对时间要求敏感。

那么，你的公司将如何从这个灾难中恢复？是否有任何计划？操作丢失将给公司收益带来多大的损失？这些操作多快可以恢复？这些都可能促使你的管理团队在灾难恢复上进行投资。 414

另外一个原因是，如果你做好最坏的打算，并拥有一套系统恢复流程，就更容易处理一般的中断。规划是灾难恢复的最重要组成部分，保证了业务的连续性。

没有人幸免于灾难

像水灾或火灾这样的灾难，几乎可以在任何地点、任何组织发生。有些地方更容易发生自然灾害。不论你的组织机构建立在何方，总存在一些风险因素。但是，有一种类型的灾难可能在任何地点、任何时候发生，那就是人为灾难。

所有的组织机构都需要防止恶意的人为活动，包括由内部人员执行的恶意操作。即使系统的某些部分得到了全面的保护，有些部分还是可能会被少数不安全的网站干扰。

例如，假设你的公司所在地很少发生自然灾害，远离地震断裂带，也没有什么水灾风险。公司的建筑配有优良的灭火系统，并且保护良好，还有训练有素的应对人员进行全天候的监控。总之，由于自然灾害引起的物理性损失的风险很低。前人已经想好了一切，管理团队觉得不需要灾难恢复计划。

现在，假设某个雇员想从内部破坏组织。在设计灾难恢复计划的时候，问问自己："一

个被信任的员工会对系统、数据或基础设施造成什么样的破坏？”虽然这样想看起来令人不安，甚至有些偏执，看看那些由内部的蓄意破坏产生的报告，便会让你觉得无法想象——你的雇员可能是破坏者。

良好的灾难恢复计划包括减少风险破坏的措施，这些措施包括识别潜在的漏洞，例如访问建筑和设备，以及查看敏感系统和数据的管理权限等。

灾难恢复的目标是尽快地重建组织的运营能力，这意味着必须恢复其信息技术。灾难恢复的最佳实践包括风险评估和各种信息技术恢复的规划——包括物理技术和电子技术。良好的灾难规划包括重新确定组织的物理位置，以及重新构建信息技术。还要包括通过权宜之计获得任何设备的能力，这些方式包括使用备用设备、用其他站点的设备取代失败站点的设备或者购买新的设备。

415 从头开始重建网络和数据需要使用最少的计算和网络设备，将数据和应用恢复到可接受的操作水平。因此，需要确定至少使用哪些技术，使得组织的运营收益损失较少或没有损失。

人员也是规划过程中必须考虑的关键资产之一。因此，在紧急事件发生的时候恢复程序会通知所有人员。可能只需要一系列简单的电话就可以办到，每个员工负责向一组同事传递重要信息。也可能是一个复杂的自动联系系统，向所有雇员发送预先记录好的信息。大多数自动系统通过员工的确认信息来进行评估。最好的通知系统还支持多点升级，例如，首先电话通知员工的家庭，接着是电子邮件、移动电话和短信息等。

灾难恢复还需要政策支持，包括紧急事件的风险评估（什么资产处于危险中，每种资产有多重要），以及如果命令层有人不在（或者去世）由谁来做决定。例如，如果网络负责人员不在，就指定其下属承担相应的责任（或进行必要的决策）。

灾难恢复规划

规划的内容很广泛，包括所有已知的或预期情景的应急方案。例如，一个良好的灾难恢复规划有一套书面程序说明如何重建站点，以及从头开始建立信息系统的步骤说明。

我们将指出某些灾难恢复规划的一个致命的缺陷。花费无限的时间和资源制定灾难恢复计划，然后存到系统硬盘上，这样做将是白费力气。请花点时间保护好你的灾难恢复计划副本吧。

灾难恢复工作流

目前你可能还有几个疑问，如何开始灾难恢复？ 怎样调查信息技术并交给老板一份灾难

恢复计划？一切都从目标考查开始，从目标开始进行计划，最终就会形成自己的灾难恢复文档。

听起来这需要大量工作，事实上确实如此。大多数组织机构成立了跨组织的专业团队。最成功的团队是由这样的人组成的：他们知道危机是什么（组织可能破产，且每个人都可能失去工作），并且他们的专业知识足以识别系统的不稳定性。

显然，你还必须做好这样的准备：那些建立、测试和制定灾难恢复计划的人可能不是该计划的执行人员。这强调了以下两点的重要性：全面而明确的文档，以及重复而清晰的责任划分。 416

下面只是简单地描述了常见灾难恢复计划的步骤，后面的章节会给出详细介绍：

1. 建立你的灾难恢复小组。假设组织中有很多员工，那么现在就告诉经理：你需要一个团队来完成灾难恢复计划。确定该团队的核心角色，保证团队中包含各个部门的人员，避免只有技术熟练的人。你需要总览整个组织机构，要做到这一点，只有使团队成员多样化。

2. 制定任务声明。研究系统的当前操作状态，明确组织可接受的最低操作水平，确定灾难恢复过程的目标。

3. 让管理层介入。除非该任务是管理层发起的，否则你有责任说服你的管理层加入，因为灾难恢复需要他们的支持才能成功。这可能是获得所需资源（时间、预算、人员等）完成目标的唯一方法。

4. 为最坏的结果做准备。很多人觉得这一条最有趣。你应该记录以下情况：本地区可能发生的灾难性事件的类型，包括所有来自于盗窃、蓄意破坏、气候事件、火灾，甚至更糟的灾难事件。务必从这些情况开始记录文档，因为后面开发恢复计划时需要用到这些信息。

5. 评估库存及组织资源。为组织的所有信息技术列个清单。务必包含系统成功运作所需的所有技术，其中包括员工工作站和网络连接。你应该列出目前存在的东西，而不是你认为的最小集合。将资产减少到最小集合是后面需要做的事情。还需要记录目前的组织结构图和指挥链，并标出主要和次要决策者。所有这些都要记录并保存。

6. 进行风险评估。确定每一项资源损失可能对组织造成的影响，这将形成系统的最低操作水平。应该为每个资源设定优先级，甚至建立多个可接受的操作级别。一定要谨慎研究应用程序及其需要的数据，这将有助于决定事件响应的程序。你可能只需要部分恢复，例如，为了节约时间和资金，只是重新部署剩余的系统，而不是建立 417

一个全新的系统来进行灾难恢复。这样你可以在较低的性能下短时运营。

7. 制定应急计划。你已经了解了灾难场景，熟悉了库存资产，并进行了风险评估，现在可以开始起草灾难恢复计划了。应急计划可以采取任何你觉得有用的形式，但是大多数计划者采用列表和叙述的形式制订流程。灾难计划书写格式不限，只要团队认可即可。

8. 建立验证程序。一旦计划在执行过程中失败，那么计划将毫无用处。现在需要制定灾难恢复计划的验证程序。从为每个应急事件制订测试实例开始，然后执行需要的调整。回到步骤 4，循环后面的步骤，直到完善了灾难恢复计划为止。

9. 熟能生巧。精炼了灾难恢复计划之后，开始实战练习，以确保计划可行。最终的目的是向管理层展示可以实现各个级别的运营水平。否则，回到步骤 6，循环后面的步骤，直到得到一套完整且可重复的流程。



灾难恢复计划应该每年至少进行一次完整的操作测试，或者在组织或信息技术发生重要变化时做完整的操作测试。

灾难恢复工具

虽然灾难恢复不仅仅包括数据和计算机，但是灾难恢复的重点常常是数据和信息技术能力。有很多工具和策略可以用于灾难恢复计划。下面介绍一些常见的工具和策略。

- 备用电源

 通常包括某些不间断供电形式。这类设备的成本和耐用性在很大程度上取决于公司允许的停机时间。如果要求必须连续运营，则需要一套可以长时间为设备供电的系统。
- 网络连接

 根据你的需要通过电子方式与客户和业务伙伴交流，你可能需要考虑冗余的或其他的网络访问。这可能简单到仅通过使用不同的媒体（如再加一条连接到访问点的光纤）来实现，也可能复杂到需要使用其他替代的连接点（如卫星或移动设备）。
- 替代站点

 如果你的公司必须具有完整的能力且尽可能少停机，你可能需要确保在没有任何数据或操作丢失的情况下迅速恢复系统。最后的办法是使用替代站点安置你的信息技术甚至整个公司。这个替代站点形成一个移动办公室（临时办公室拖车或类似平台），它包含了你的信息技术的副本。
- 替补人员

 灾难恢复计划常常忽视这部分。一个必须考虑的可能的灾难是损失部分甚至全部关

418

键工作人员，例如，传染病、被敌对公司挖走甚至是死亡。不论何种情况，都应该对组织机构内的重要角色进行分析，并提出灾难事件中的替补措施。显然可以选择交叉培训员工。这样可以丰富你的人才库，还可以提高员工价值。

- 备份硬件
 可能连续运营最明显的需求就是额外的硬件。高可用性技术可以部分满足这种需求，但最终还是要在站点外（或在替换站点上）存储备用硬件，这样才能将它们快速地投入服务中。这种硬件绝不能只是摆设，而应该保持更新并定期运行。
- 安全的保险库
 计划者容易忽视的另一个方面是在安全的地方存储重要数据的备份。应该准备一个安全的存储空间（安全级别取决于你的数据的价值和敏感度）以匹配你的停机需求。也就是说，如果要求停机时间很短，那么银行或类似的库可能无法快速获取数据，而将关键数据备份存储在地下室或公共场所又可能不够安全。
- 高可用性
 如前所述，高可用性选项形成了灾难恢复计划的第一层防御。将失败转移到其他数据存储库可以帮助你应付一些小事件。



始终将你的数据和灾难恢复计划的副本保存在站点外的安全位置。

所有灾难恢复工具和策略的一个经验法则是：需要从灾难中恢复的速度越快，需要恢复的操作能力越强，那么你付出的成本将越高。这也是开发具备多个级别的运行能力的系统很重要的原因——那样你可以调整计划，并且可以在不同层次的损失和环境下进行管理。有可能你的公司遭遇经济衰退或其他财政限制，不得不牺牲某些运行能力。考虑到的情况越多，灾难恢复计划就越强大。 419

数据恢复的重要性

大多数企业把数据看成是最有价值的资产（超过员工的价值，也是最不容易被代替的资产），所以大多数灾难恢复计划的关注点是恢复数据，并使这些数据可用。这些都是数据问题。

数据恢复是指将组织机构的数据恢复到可接受的运行状态的一系列计划、原则和流程。数据恢复最重要的就是数据的备份和恢复能力。成功的备份和恢复能力有时也被称为弹性（resiliency）或可恢复性（recoverability）。

数据恢复主要是计划、执行和审计数据备份，以及恢复数据操作。其中审计经常被忽略。

一个优秀的备份和恢复服务应该是可依赖的。发生数据丢失、损坏或销毁的备份系统常常失效。这通常表明有地方出错了，并试图恢复之。显然，如果数据不存在或在灾难时刻被损坏，将导致数据恢复很困难。

一些术语

几乎所有的数据恢复和备份解决方案都会用到这两个术语，所以了解它们是很重要的：

- 恢复点目标（RPO）
 可接受的系统状态。因此，RPO 代表了服务（运行能力）下降的最大可接受程度或数据丢失的最大可接受程度。使用这些标准衡量恢复操作是否成功。
- 恢复时间目标（RTO）
 能够容忍损失的最大时间，又称停机时间。

RTO 在很大程度上受 RPO 影响。如果需要恢复所有数据（或几乎所有数据，如复制过程中的所有事务），则恢复过程更麻烦些，因而想做到快速恢复即低 RTO，需要付出的成本将更高。较高的 RPO 或较低的 RTO 需要更高的软硬件投资，还需要培训员工使用该系统才能保证。RTO 较低和 RPO 较高，因为这样的系统通常都很复杂。例如，如果想要 RTO 低于 3 秒且 RPO 能够保证无数据丢失，则必须制订一个可扩展的高可用性解决方案，该方案可以容忍在服务器及数据存储等完全丢失的情况下，无数据（或时间）丢失。

420 备份和恢复

本节介绍了两种重要的数据恢复工具，还介绍了备份和恢复的概念、需要执行的计划，以及可行的备份和恢复的解决方案。

对于像 MySQL 这样的数据库，备份和恢复功能是指复制数据副本，存储这些副本然后重新加载，使数据回到备份时间点的状态。

下面向不太熟悉备份和恢复系统及其 MySQL 的可行方案的人们进行相关介绍。我们将探索各种解决方案的好处，以及如何创建存档计划，使得恢复数据的损失最小。

为什么要备份

有些人可能认为如果使用了复制或某种形式的硬件冗余，就没有必要进行备份了。虽然对于机械或电子故障事件中数据的快速恢复来说，这种说法或许是合理的，但是如果发生灾难性事件，这些措施将无济于事。

表 12-1 描述了最有可能丢失数据的情况，并给出了各种情况的恢复方法。可以看到，大

部分情况都或多或少地受益于备份。除了可以从失败和错误中进行恢复以外，还可以将备份纳入日常数据保护计划中。这么做的理由很充分，你可以为复制拓扑添加 Slave 服务器，将其用做故障转移工具，甚至作为网络之间传输数据的一种方式。

表12-1 ：常见的数据丢失原因

错误类型	描述	恢复方法
用户错误	用户无意删除数据或将数据更新为不正确的值	从备份中恢复删除的数据
断电	一个或多个系统断电	使用不间断的供电系统
硬件故障	一个或多个系统或组件故障	使用冗余系统或复制数据
软件故障	数据在转换时被改变或丢失	该类故障可能很难侦查，需要从备份中恢复数据（如果没有办法修复转换操作）
基础设施问题	放置设备的基础设施无法居住，且数据连接丢失	可能需要一套新的基础设施，用于建立一个可以接受的运行系统
网络故障	装载数据的服务不能访问	重新连接或建立一个新的数据储存库
蓄意破坏	数据被有意偷走或损坏	关闭安全漏洞，并审查数据

如果开发了自己的日常备份计划并使用了异地存储，那么你可能已经做好了足够的灾难恢复准备。例如，假设数据库系统的冗余硬件，甚至复制服务器（即 MySQL 复制中的 Slave）突然消失，或者被盗，或者发生了灾难事故，这时将发生什么？异地站点的定期备份允许将数据或数据库系统恢复到最后一次备份的状态。

421

硬件恐怖事故

表 12-1 描述了数据丢失的常见分类。硬件故障不经常发生，但是一旦，发生远比表 12-1 描述的严重得多。下面给出现实中可能发生的硬件故障，以及相应的恢复办法。

- 丢失硬件阵列中的多个磁盘
 如果一个硬件 RAID 的多个磁盘发生故障，很可能无法恢复。如果发生这种情况（发生的可能性往往比你想象的大），那么别无选择，只有从备份阵列中恢复数据。
- Master 和 Slave 上的磁盘都发生故障
 如果你的 Master 和 Slave 系统都发生故障（尤其是发生磁盘故障），那么你将丢失热备份（hot standby）。这种情况有多种表现形式，例如，某个特定的表（或分片，shard）或者其所在的磁盘可能同时在 Master 和 Slave 上被损坏。这时恢复数据可能是唯一的解决办法。

- 备用电池发生故障
 这也许是最丢脸的状况。高效的供电备份系统在你需要它的时候（在停电时期）发生了故障。如果你需要为停电做准备，请考虑为主电源备份故障建立一个备份计划。

现在假设你的数据库系统运行在商业硬件上（MySQL 因独立于硬件而出名）。要想实现备份，你可以购置新硬件或重用已有硬件，迅速使数据库系统恢复运行（当然，这取决于需要恢复的数据量）。

关键在于，日常数据备份是一项基本且完备的数据恢复实践。高可用性如复制或 RAID 硬件对秒级恢复来说显然更好，但是它们无法处理灾难性损失。我们已经知道复制可以防止数据丢失，但是如果 Master 和 Slave 都损坏了且不可修复，那又怎么办呢？在这种情况下，只有有效的最新备份可以拯救你。

以下各节将详细讨论备份，并展示如何创建 MySQL 数据的备份。

422

备份的期望

备份以某种可恢复的方式创建数据的副本。此外，备份副本必须保持一致。对于事务性数据库，这意味着备份只包含备份数据之前提交的事务，不包括部分提交或未提交的数据。备份还应该支持监控，用于验证备份的性能和数据状态。

有以下几种数据备份形式：

- 完全备份
 对服务器进行完整备份，没有任何遗漏。该种形式周期最长，占用的存储空间最大。
- 差异备份
 仅对上一次完全备份之后发生变化的数据进行备份。一般来说，该种备份比完全备份需要的空间少，而且速度也较快。
- 增量备份
 只备份自上次增量备份或完全备份后发生变化的数据。通常需要某种形式的更改日志（如 MySQL 中的二进制日志）。该类备份通常比差异备份或完全备份周期更短，另外，根据自上次备份以来发生的变化量的大小，所需的备份空间可能很少。

制订数据恢复计划时，需要为备份文档（也被称为备份映像或备份文件）命名。例如，用日期加上其他相关信息为文件命名，如 *full_backup_2009_09_09* 或 *incr_backup_week_3_september*。命名没有什么限制，可以根据需要使用任何你认为有意义的名字。

还原过程的期望

还原操作用备份文档中的数据替换系统中的数据，使得被替换的数据和文档中的数据相同。像备份一样，还原进程必须支持监控，以验证还原的性能和数据的状态。

不幸的是，很少有备份系统完全满足这些备份和恢复的准则。满足条件的通常是那些专业平台（使用定制的硬件和软件），这样的平台价格昂贵且很难维护。本章介绍了备份和还原 MySQL 数据的经济方案。

逻辑备份与物理备份

一个容易被误解的备份概念是逻辑备份和物理备份之间的区别。所选择的备份模式可能影响备份和恢复数据的效率。

逻辑备份（*logical backup*）仅仅是一些普通 SQL SELECT 查询的集合。通常通过表扫描（即逐条记录遍历数据）创建逻辑备份。 423

物理备份（*physical backup*）是指原始二进制数据（文件）的副本，这些副本通常是操作系统级别的文件。任何复制数据、索引和缓冲内存（文件）且不是逐条记录访问的备份方法都为物理备份。

可以想象，逻辑备份比物理备份慢得多。这是因为系统必须使用普通的 SQL 内部机制一次读取一个记录，而物理备份通常使用操作系统功能，没有那么多的开销。但是，物理备份可能需要锁定表及其相关的二进制文件，直到整个复制过程结束；而某些逻辑备份在运行时不会锁定表或阻塞表的访问。

选择使用逻辑备份还是物理备份比你想象的要难。例如，如果想要创建包含所有数据的 MySQL 数据库服务器的副本，可以简单地将数据库离线，并关闭服务器，然后将整个 *mysql* 目录复制到另外一个计算机上。如果更改了默认路径，还必须复制 InnoDB 文件的数据目录。这样就建立了与第一个实例相同的服务器实例。这可能是建立复制拓扑的好方法，但是对于那些无法离线的重要数据库的备份来说，这样会很不方便。此时，逻辑备份可能是最好的（而且是唯一的）选择。

制定归档计划

使用备份和还原工具需要遵循一定的原则，还有一些注意事项。最容易忽略的步骤是制订归档计划，这将决定你备份数据的频率。

首先问问自己："我可以承受多少数据损失，然后从头恢复它们？"也就是说，如果数据全部丢失，你最多可以忽略多少数据？或者说你能够承受多少数据无法恢复？这就是你的恢复点目标（RPO），即进行业务运营必备的操作能力水平。

很明显，任何数据丢失都是坏事情。你必须在规划 RPO 时考虑数据本身的价值。你可能发现某些数据比其他数据更有价值。某些数据可能对公司的福利和稳定很重要，而其他的数据或许没那么重要。显然，如果你确定某些数据对组织机构很重要，那么必须保证它们能够完全恢复。因此，你可能会想出若干不同的 RPO。一般情况下，最好先制定几个级别的 RPO，然后确定哪个更适合恢复需求。

制订归档计划很重要，因为它决定了备份频率及备份能力（即备份数据的多少）。如果
424 你不能容忍任何数据丢失，更加需要扩展高可用性选项以支持异地备份某些数据。但是即使那样做也不能保证万无一失。例如，Master 中的损坏或蓄意破坏可能波及其他副本，并在一段时间内未被发现。

通常采用恢复丢失数据所能容忍的最大时间来衡量对数据丢失的容忍程度。也就是说，在数据恢复过程中，你可以承受数据多长时间不可用？这直接说明了停机过程中你的组织机构可以承受丢失多少钱。对于有些组织机构来说，不能访问数据的每一秒钟的代价都是很高的。另外，某些数据比其他数据对公司盈利更有价值，这些数据的恢复需要花费更多的时间。

除此以外，还应该制定一些时间期限，根据当前业务的状态决定使用哪个时间期限，这些时间期限形成了恢复时间目标（RTO），与相应的恢复点目标（RPO）级别匹配，就可以知道不同时间期限的恢复过程需要多长时间。确定 RTO 级别需要将数据重新导入系统，或者重做某些工作以重新获得数据（例如，再次下载数据或从商业伙伴那里获取更新数据）。

确定了能够容忍的数据损失量（RPO）及其恢复时间（RTO）后，检查备份系统的能力，并选择能够满足需求的备份频率和方法。但还不止这些。还应该建立自动化任务，使这些任务在最有益（或损失最小）的时候自动执行备份。

最后，通过试验数据恢复（如还原数据）定期测试备份，确保备份的完备无缺和可实施性，从而保证在低风险下进行安全的数据恢复。

备份实用程序和操作系统级的解决方案

Oracle 提供了备份 InnoDB 和 MyISAM 表的解决方案 InnoDB Hot Backup。此外，还有很多第三方方案用于执行 MySQL 上的备份。你也可以在操作系统级别执行一些不成熟的但是有效的备份。

本章最后一节简要讨论了其他流行的备份解决方案：

- InnoDB Hot Backup；

- 物理文件复制；
- `mysqldump` 实用工具；
- XtraBackup；
- Logical Volume Manager 快照。

还有其他选择，但是大多数与这些方案相似。

InnoDB Hot Backup应用

如果仅使用 InnoDB 作为存储引擎，或者只定期备份 InnoDB 表而不备份其他类型的表，那么可以使用 Oracle 的 InnoDB Hot Backup 应用。InnoDB Hot Backup 是个商业产品，最大优点是可以在备份的时候保持数据库的查询和更新继续运行，而无须让服务器离线或显式锁定表。



Hot Backup还可以备份MyISAM表，但是需要在备份过程中阻塞MyISAM表的更新。

InnoDB Hot Backup 在不停止任何事务的情况下创建了所有数据库的一致性副本。可以备份部分或全部数据库，压缩备份的输出文件，还可以进行选择性备份（例如仅备份指定的表）。

InnoDB Hot Backup 在 MySQL 4.0 及更高版本上可用，支持 Linux、UNIX 和 Windows 平台。如果你有兴趣学习更多关于 InnoDB Hot Backup 的知识，请参看 InnoDB Hot Backup 网站。



Oracle 曾在 2010 年 MySQL 用户大会上宣布 InnoDB Hot Backup 的下一个版本更名为 MySQL Enterprise Backup，该产品计划提供许多新的功能。

InnoDB Hot Backup 解决方案由两个文件组成：*ibbackup* 和 *innobackup*。在下面的章节中，将详细介绍 InnoDB Hot Backup 应用，并演示如何执行数据备份和数据恢复。

其核心备份工具是 *ibbackup*。它可以执行三种主要操作：备份数据、应用备份日志和还原数据。Perl 脚本 *innobackup* 为 *ibbackup* 提供了高级接口及一些扩展功能。

使用ibbackup备份

ibbackup 的使用有点不寻常：必须在命令行中指定两个备份所需的文件作为参数。第一

个文件通常是标准的 *my.cnf* 配置文件，包含需要备份的数据库的相关信息。第二个文件包含用于保存备份的文件信息。每个文件的指令名称相同，指令的值指向存储数据库的文件系统中的文件信息，该命令常用的执行形式如下所示：

```
ibbackup my.cnf backup.cnf
```

426 每个文件中的参数是：

```
datadir = directory
innodb_data_home_dir = directory
innodb_data_file_path = parameter-list
innodb_log_group_home_dir = directory
innodb_log_files_in_group = group_number
innodb_log_file_size = size
```

可以在 MySQL 的在线参考文档中找到这些参数的定义，下面给出了第一个配置文件的典型配置信息，包含了需要备份的数据：

```
[mysqld]
datadir = /usr/local/mysql/data
innodb_data_home_dir = /usr/local/mysql/data
innodb_data_file_path = ibdata1:10M:autoextend
innodb_log_group_home_dir = /usr/local/mysql/data
set-variable innodb_log_files_in_group = 2
set-variable innodb_log_file_size = 20M
```

假设要将备份文件写入 /home/cbell/backup 目录，那么。在 `ibbackup` 命令中被称为 *backup.cnf* 的结果配置文件如下所示：

```
datadir = /home/cbell/backup
innodb_data_home_dir = /home/cbell/backup
innodb_data_file_path = ibdata1:10M:autoextend
innodb_log_group_home_dir = /home/cbell/backup
set-variable innodb_log_files_in_group = 2
set-variable innodb_log_file_size = 20M
```

还可以使用 `--compress` 选项压缩备份：

```
ibbackup --compress my.cnf backup.cnf
```

应用备份日志

由于备份过程中 InnoDB 数据库仍在运行，因此无法保证所有时间点的数据都是一致的。如果想得到一致的备份，必须将当前日志应用到备份文件中以同步数据。*ibbackup* 工具可以实现这一点，它创建了一个 *ibbackup_logfile* 日志，其中包含了备份过程中发生的数据更改。将这个日志文件应用到备份文件中，然后回滚到备份完成时对应的时间点上。

执行此操作只需要备份配置文件和 --applylog 选项，即：

```
ibbackup --apply-log /home/cbell/backup.cnf
```

如果备份时使用了 compress 选项，那么还要提供 --uncompress 选项，将日志应用到压缩备份中，如下所示：

```
ibbackup --apply-log --uncompress /home/cbell/backup.cnf
```

使用ibbackup还原数据

427

恢复数据需要上一节所描述的应用日志。这个操作其实是指将 MySQL 实例指向数据库的备份副本。它不会将文件复制到 MySQL *datadir* 目录。如果需要，必须手动复制文件或使用 *innobackup* 脚本。因为 *ibbackup* 工具不会覆盖任何数据。使用下面的命令启动 MySQL 实例并使用数据的备份：

```
mysqld -defaults-file=/home/cbell/backup.cnf
```

innobackup脚本

innobackup 文件是一个 Perl 脚本，可以自动执行 *ibbackup* 的许多操作，例如创建备份，还原数据，使用备份中的数据启动 MySQL 实例，或将备份目录中的数据、索引和日志文件复制到其原始位置。



innobackup 脚本目前在 Windows 平台上不可用。

与 *ibbackup* 程序有所不同，*innobackup* 不需要在独立的配置文件中指定命令，而是使用命令行选项指定备份或恢复的参数。表 12-2 中描述了这些选项。

表12-2：Innobackup选项

选项	功能
--help	显示所有选项的列表
--version	显示脚本的版本
--apply-log	将备份日志应用到备份中，并为带有备份文件的 MySQL 服务器的启动做准备
--copy-back	从备份文件中复制数据和索引文件到它们原始位置
--use-memory=MB	传递给 ibbackup，该选项在数据还原中控制内存的使用量
--sleep=MS	传递给 ibbackup，该选项使程序在每复制 1MB 数据后停止一下
--compress=LEVEL	传递给 ibbackup，该选项提供压缩级别供使用

续表

选项	功能
--include=REGEXP	传递给 ibbackup，该选项指导程序只备份那些与正则表达式匹配的表文件，应用于选择性备份
--uncompress	传递给 ibbackup，该选项解压压缩备份
--user=NAME	连接到服务器的用户名称
--password=PWD	用户密码
--port=PORT	服务端口
--socket=SOCK	服务的 Socket

使用innobackup执行备份

创建备份只需要两个选项：服务器的配置文件和备份文件的存储位置。

```
perl innobackup /etc/mysql/my.cnf /home/cbell/backup
```

如果你需要一致性备份，还需要指定 --apply-log 选项：

```
perl innobackup --apply-log /etc/mysql/my.cnf /home/cbell/backup
```

使用innobackup恢复数据

为了还原数据，需要应用日志，然后使用 --copy-back 选项将文件复制到其原始位置。下面给出了这些命令的实例。复制之前必须停止服务器，之后再重新启动。

```
perl innobackup --apply-log /etc/mysql/my.cnf /home/cbell/backup
mysqladmin -uroot shutdown
perl innobackup --copy-file /etc/mysql/my.cnf /home/cbell/backup
/etc/init.d/mysql start
```

其他特性

InnoDB Hot Backup 还支持 PITR（即时恢复）。想了解更多，请参考 InnoDB Hot Backup 文档（*http://www.innodb.com/doc/hot_backup/manual.html*）。



可以下载 InnoDB Hot Backup 的试用版，试用期是 30 天，在此期间，可以无限制使用 InnoDB Hot Backup。要注册免费的试用版，请登录 *http://www.innodb.com/products/hot-backup/order/order/*。

物理文件的复制

最简单和最基础的 MySQL 备份的形式是简单文件复制。不幸的是，为了得到更好的结果，这种方法需要停止服务器。要执行文件复制，只需停止服务器，然后复制服务器上的数

据目录和所有安装文件。一种常见的方法是使用 UNIX `tar` 命令创建归档，然后将这个归档文件移到另一个系统上，并恢复数据目录。

下面是一个典型的 `tar` 命令，备份一个数据库服务器上的数据，然后在另一个系统上恢复这些数据。在需要备份的服务器上执行以下命令，其中 *backup_2009_09_09.tar.gz* 是要创建的文件，*/usr/loca/mysql/data* 是数据目录的路径：

```
tar -czf backup_2009_09_09.tar.gz /usr/loca/mysql/data/*
```

backup_2009_09_09.tar.gz 文件所在的文件夹由两个服务器共享（或者你必须手动将它复制到新的服务器上）。现在，在你需要恢复数据的服务器上，打开新安装的 MySQL 数据 429
库的根目录，如果数据目录存在，删除现有的数据目录，然后执行以下命令：

```
tar -xvf ../backup_2009_09_09.tar.gz
```



前面讲过，最好使用有意义的文件名为备份映像命名。

从这个例子可以看出，在操作系统级备份数据是很容易的事情。你不仅得到了一个便于迁移的压缩归档文档，还可以从快速文本复制中获益。通过简单地复制数据目录中的单个文件或子目录，甚至可以执行选择性备份。

遗憾的是，`tar` 命令仅适用于 Linux 和 UNIX 平台。如果你在 Windows 平台上安装了 Cygwin，并且 Cygwin 包含了 `tar` 命令，那么 Windows 平台上也可以使用 `tar` 命令。

除了使用 Cygwin 和其他的 Unix-on-Windows 包以外，Windows 中最接近 `tar` 命令的是资源管理器的文件夹归档功能或像 WinZip 这样的归档程序。打开 Windows 资源管理器，然后进入你的数据目录。但不是直接打开目录，而是右键单击数据目录，并选择压缩数据的选项，即选择 SendTo → Compressed (zipped) Folder，然后为 *.zip* 文件提供一个名称。

虽然物理文件复制是最快、最简单的备份形式，但是它要求关闭服务器。然而，如果你能保证在文件复制过程中没有数据更新，就不需要关闭服务器。为此，必须锁定所有表，并执行 `flush tables` 命令，然后在复制文件之前将服务器离线。



这与克隆 Slave 的过程类似。具体如何使用文件复制克隆 Slave 详见第 2 章。

此外，根据数据的大小而定，服务器不仅必须在复制文件时脱机运行，而且在以下情况

下也必须脱机运行：任何额外数据加载时（如加载缓存项）、快速查找下使用内存表时等。因此，物理复制备份可能不适合某些安装项。

幸运的是，Tim Bunce 创建了一个名为 *mysqlhotcopy.sh* 的 Perl 脚本，来自动运行这个过程。它位于 MySQL 安装文件目录的 *./scripts* 文件夹下。它允许对数据库进行热备份。但是该脚本只能用来备份 MyISAM 或文档存储引擎，且只能在 UNIX 和 Netware 操作系统上运行。

430 *mysqlhotcopy.sh* 工具还有一些自定义功能。可以在 *http://dev.mysql.com/doc/refman/5.4/en/mysqlhotcopy.html* 上找到更多关于它的信息。

mysqldump工具

物理文件复制功能最常见的替代者就是 `mysqldump` 客户端应用。它是 MySQL 安装的一部分，最初是由 Igor Romanenko 捐献给 MySQL 的。`mysqldump` 创建了一组 SQL 语句集，重新运行这些 SQL 语句可以重建数据库。例如，运行备份时，输出结果包含了所有创建数据库及其表所需的 `CREATE` 语句，以及所有重建这些表数据的 `INSERT` 语句。

如果需要在数据文本上执行查找和替换操作，使用 `mysqldump` 可以很方便地完成该操作。只需备份数据库，使用文本编辑器编辑结果文件，然后恢复数据库以使更改生效。许多 MySQL 用户使用这种方法来更正所有由批量编辑而导致的错误。这比使用复杂的 `WHERE` 子句写 1000 个 `UPDATE` 语句容易多了。

使用 `mysqldump` 的缺点是：它比文件级（物理）备份的二进制文件复制（如 InnoDB Hot Backup、LVM 或简单的脱机文件复制）花费的时间要长，而且需要更大的存储空间。如果需要经常进行备份，并希望在系统故障后快速恢复，或者需要通过网络传输备份文件，那么这个时间成本将会非常大。

使用 `mysqldump` 可以备份所有数据库、特定的数据库集合，甚至还可以备份指定数据库中的某些表，如下所示：

```
mysqldump -uroot -all-databases
mysqldump -uroot db1, db2
mysqldump -uroot my_db t1
```

还可以使用 `mysqldump` 执行 InnoDB 表的热备份。`--single-transaction` 选项在备份开始的时候发出 `BEGINE` 语句，指示 InnoDB 存储引擎以一致性读的方式读取表。因此，你做的任何更改都会应用到表上，但是数据在备份时被冻结。然而连接不能使用 DDL（数据定义语言）语句，如 `ALTER TABLE`、`DROP TABLE`、`RENAME TABLE`、`TRUNCATE TABLE`。这是因为一致性读无法隔离 DDL 更改。



--single-transaction 选项和 --lock-tables 选项是相互排斥的，因为 LOCK TABLES 采用的是隐式提交。

mysqldump 工具有很多用于控制备份的选项，表 12-3 描述了一些比较重要的选项。完整的选项集合参见 *http://dev.mysql.com/doc/refman/5.4/en/mysqldump.html* 的 MySQL 在线参考手册。 431

表12-3：mysqlbackup选项

选项	作用
--add-drop-database	在每个数据库前包含一个 DROP DATABASE 语句
--add-drop-table	在每个表之前包含一个 DROP TABLE 语句
--add-locks	在被包含的表的前面加上 LOCK TABLES 后面加上 UNLOCK TABLES
--all-databases	包含所有的数据库
--create-options	在 CREATE TABLE 语句中包含所有的 MySQL-specific 表选项
--databases	只包含数据库的列表
--delete-master-logs	在 Master 上，在执行备份后删除二进制日志
--events	备份被包含的数据库中的事件
--extend insert	另一种将所有的记录作为 VALUES 子句的 INSERT 语法
--flush-logs	在开始备份前刷新日志文件
--flush-privileges	在备份 *mysql* 数据库后包含一个 FLUSH PRIVILEGES 语句
--ignore-table=db.tbl	不备份指定的表
--lock-all-tables	在 dump 中锁定所有数据库中的所有表
--lock-tables	在包含表前锁定所有表
--log-error=filename	将警告和错误追加到指定文件中
--master-data[=value]	在输出结果中包含二进制日志文件名及其位置
--no-data	不写入任何表的行信息（只包含 CREATE 语句）
--password[=password]	连接到服务器的密码
--port=port_num	连接时使用的 TCP/IP 端口号
--result-file=filename	结果输出到指定文件中
--routines	包括存储程序（过程和函数）
--single-transaction	在数据 Slave 中导出之前执行 BEGIN SQL 语句，这允许 InnoDB 表的一致性快照
--tables	重写 --databases 选项
--triggers	包含触发器

续表

选项	作用
--where='condition'	只包含在 conditon 中被选中的行
--xml	生成 XML 输出结果

432



还可以在 MySQL 配置文件中的 [mysqldump] 下使用这些选项。在大多数情况下，只要删除行首的破折号就可以指定选项。例如，要产生 XML 输出，需要在配置文件中包含 xml 选项。

mysqldump 的一个非常方便的功能是转储数据库模式。通常可以这样做：使用一组 CREATE 命令（不包括 INSERT 语句）来重新创建所有的对象。这种用法有利于保存模式更改的历史记录。如果使用 --no-data 选项，加上其他包含所有对象的选项（如 --routines，--triggers），可以使用 mysqldump 创建数据库模式。

请注意 --master-data 选项，该选项有助于执行 PITR，因为它保存了二进制日志信息，与 InnoDB Hot Backup 一样。

还有很多选项允许你控制该工具的运行。如果使用 SQL 语句创建备份是你的最佳选择，那么可以研究一下其他选项，以更好地使用 mysqldump。

XtraBackup

独立开源供应商 Percona 是 MySQL（实际上是 LAMP）领域的专业咨询公司。它创建了一个名为 XtraDB 的存储引擎，该存储引擎是基于 InnoDB 存储引擎的开源存储引擎。XtraDB 改进了 InnoDB，具备更好的硬件伸缩能力，且向后兼容 InnoDB。

Percona 创建了 XtraDB 的热备份方案——XtraBackup。这个工具优化后可供 InnoDB 和 XtraDB 使用，也可以用来备份和还原 MyISAM 表。它提供了很多备份相关的特性，包括压缩和增量备份。

可以从 Launchpad 上下载源代码构建 XtraBackup，网址为 *https://launchpad.net/percona-xtrabackup*。大多数平台上都可以编译和执行 XtraBackup。它与 MysqL 5.0 和 5.1 版本兼容。

XtraBackup 的在线手册地址：*http://www.percona.com/docs/wiki/percona-xtrabackup:xtrabackup_manual*。

逻辑卷管理器快照

大多数 Linux 和某些 UNIX 系统提供了另一种强大的 MySQL 数据库的备份方法，该方法使用的技术称为逻辑卷管理器（*Logical Volume Manager*，LVM）。



微软 Windows 也有一个类似的技术叫卷影复制（*Volume Shadow Copy*）。遗憾的是，没有通用的工具用于对 LVM 中的任意分区或文件夹结构进行快照。不过，可以为整个驱动器进行快照，如果驱动器上只有数据库目录，那么该功能很有用。参看 Microsoft 的在线文档以了解更多的信息。 433

LVM 是一个磁盘子系统，不需要使用陈旧的、复杂的和不可原谅的磁盘工具，它可以帮助你容易快速地创建、删除并调整卷的大小。

备份的额外好处就是进行快照，即在不影响应用程序访问卷数据的情况下，复制活动卷。其思想是进行快照，快照的执行速度相对较快，然后备份该快照，而不是备份原始卷。LVM 中的快照技术使用这样的机制：跟踪快照开始后的一切更改，只存储更改的磁盘段。因此，快照比完全复制卷占用的空间少，而且当备份完成时，LVM 将复制快照时的所有文件。快照有效地冻结了数据。

使用 LVM 和快照来备份数据库系统的另一个好处在于你如何使用卷。最好将 MySQL 安装在一个单独的卷上，那样，所有的数据都在同一个卷上，就可以使用快照迅速创建备份。有些情况下可能使用多个逻辑卷，比如一个表空间使用一个逻辑卷，甚至不同的 MyISAM 和 InnoDB 表使用不同的逻辑卷。

LVM入门

如果你的 Linux 系统上没有安装 LVM，可以使用包管理器安装。例如，在 Ubuntu 上使用以下命令安装 LVM：

```
sudo apt-get install lvm2
```

虽然并非所有的 LVM 系统都相同，但是下面的 LVM 系统是基于典型 Debian 发行版开发的，而且在像 Ubuntu 这样的系统上运行良好。我们不打算写一本关于 LVM 的完整教程，只是让你了解使用 LVM 进行数据库备份的复杂性。关于系统支持哪种 LVM 类型请参阅你的操作系统文档，或参看网上的指导文档。

在我们开始详细讨论 LVM 之前，首先花点时间了解一下 LVM 的基本概念。LVM 是分层实现的。底层是磁盘，再上一层是分区，允许磁盘之间相互通信，再上面是物理卷，它是 LVM 的控制机制。可以向卷组中添加物理卷（卷组可以包含多个物理卷），一个卷组可以包含一个或多个逻辑卷。图 12-1 描述了文件系统、卷组、物理卷和块设备之间的关系。 434

逻辑卷可以作为一个正常挂载的文件系统，也可以作为一个快照逻辑卷。快照逻辑卷的创建是在备份中使用快照的关键。以下的章节描述了如何开始使用 LVM 和备份数据。

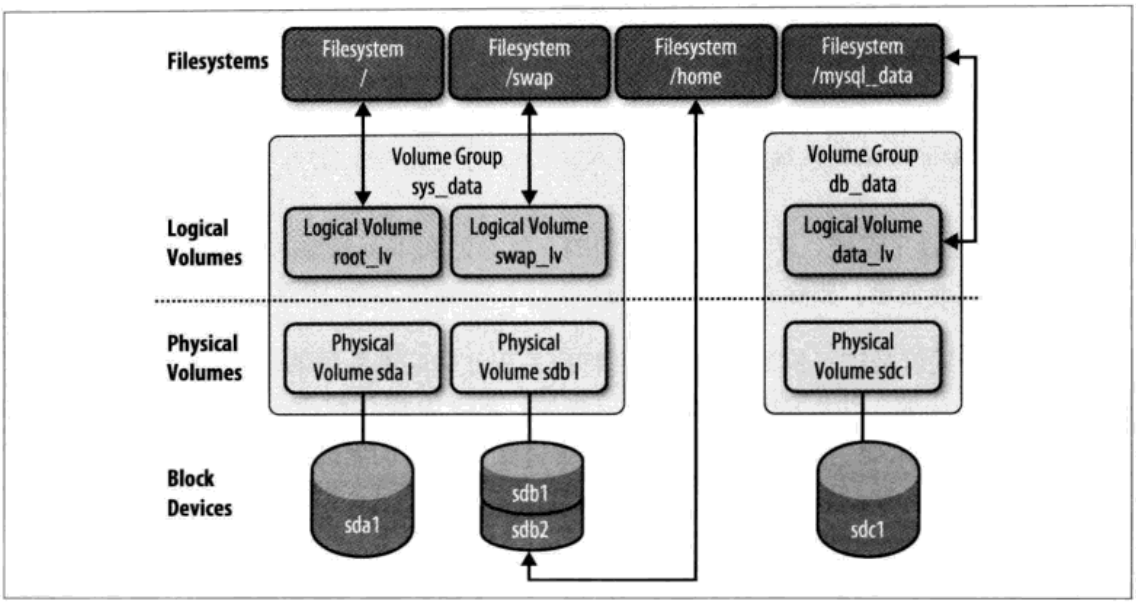


图12-1：LVM的解剖图

有几个有用的命令是你应该熟悉的。下面列出了最常用的命令及其用途，更多信息请参考相关文档。

- `pvcreate`
 创建物理卷。
- `pvscan`
 显示物理卷的详细信息。
- `vgcreate`
 创建卷组。
- `vgscan`
 显示卷组的详细信息。
- `lvcreate`
 创建逻辑卷。

435

- `lvscan`
 显示逻辑卷的详细信息。
- `lvremove`
 删除逻辑卷。
- `mount`
 挂载逻辑卷。
- `umount`
 卸载逻辑卷。

为了使用 LVM，你需要有一个新磁盘，或者可被逻辑卸载的磁盘设备，操作过程如下（这个输出结果是在安装了 Ubuntu 9.04 的笔记本电脑上产生的）：

1. 创建一个现有的 MySQL 数据目录的备份。

```
tar -czf ~/my_backups/backup.tar.gz /dev/mysql/datadir
```

2. 为驱动器分区。

```
sudo parted
select /dev/sdb
mklabel msdos
mkpart test
quit
```

3. 为驱动器创建一个物理卷。

```
sudo pvcreate /dev/sdb
```

4. 创建卷组。

```
sudo vgcreate /dev/sdb mysql
```

5. 为数据创建逻辑卷，这里我们创建了一个 20GB 的卷。

```
sudo lvcreate -L20G -ndatadir mysql
```

6. 在逻辑卷上创建一个文件系统。

```
mke2fs /dev/mysql/datadir
```

7. 加载逻辑卷。

```
sudo mkdir /mnt
sudo mount /dev/mysql/datadir /mnt
```

8. 复制文档，并在你的逻辑卷上重建数据。

```
sudo cp ~/my_backups/backup.tar.gz
sudo tar -xvf backup.tar.gz
```

9. 创建一个 MySQL 服务器实例，并使用 `--datadir` 指定逻辑卷上的文件夹。

```
./mysqld --console -uroot --datadir=/mnt
```



如果你想试用 LVM，建议使用一个允许丢失数据的磁盘。一个合适且廉价的选择是使用小型 USB 硬盘驱动。 436

以上介绍了逻辑卷的入门知识。在产品系统上开始使用 LVM 之前，请花一些时间去试用 LVM 工具，直到可以很熟练地运用它们。

备份和还原中的LVM

为了执行备份，需要刷新和临时锁定所有的表，紧接着执行快照，然后给所有表解锁。为了保证所有正在进行的事务执行结束，必须锁定表。下面给出了这个备份过程，以及相应操作的 shell 命令：

1. 在 MySQL 客户端执行 `FLUSH TABLES WITH READ LOCK` 命令。
2. 创建逻辑卷的快照（`-s` 表示快照）。

```
sudo lvcreate -L20M -s -n backup /dev/mysql/datadir
```

3. 在 MySQL 客户端执行 `UNLOCK TABLES` 命令（现在服务器可以恢复操作了）。
4. 装载快照。

```
sudo mkdir /mnts
sudo mount /dev/mysql/backup /mnts
```

5. 执行快照备份。

```
tar -[FIXTHIS]f snapshot.tar.gz /mnts
```

当然，快照的最佳用途是定期启动复制以进行下一次备份。网上有志愿者提供了自动执行上述过程的脚本，但是真正可靠的方式是删除快照然后重建，过程如下：

1. 卸载快照。

```
sudo umount /mnts
```

2. 删除快照（逻辑卷）。

```
sudo lvremove /dev/mysql/backup
```

然后重新创建快照并执行备份。如果有自己的脚本，建议在验证备份存档已经创建之后，添加一步删除快照，保证脚本执行了适当的清理操作。

如果需要恢复快照，只需恢复数据就可以了。LVM 真正的好处在于：使用 `tar` 工具创建快照和备份的所有操作，都可以通过创建定期运行的自定义脚本（如 cron job）实现，这个脚本可以帮助你自动备份。

ZFS 中的 LVM

使用 Sun 公司的 ZFS 文件系统（Solaris 10 中可用）执行备份的流程和 Linux LVM 相似，这里描述一下它们的不同。

在 ZFS 中，在池（与卷组类似）中存储逻辑卷（Sun 将可读写的卷称为文件系统，而只读的称为快照）。创建备份或副本只需创建文件系统的快照。

使用下面的命令创建可管理的 ZFS 文件系统：

```
zpool create -f mypool c0d0s5
zfs create mypool/mydata
```

使用以下命令创建备份（执行新文件系统的快照）：

```
zfs snapshot mypool/mydata@backup_12_Dec_2009
```

使用以下命令将文件系统恢复到某个指定的备份：

```
cd /mypool/mydata
zfs rollback mypool/mydata@backup_12_Dec_2009
```

ZFS 不仅提供了完全卷（文件系统）备份，还支持选择性文件恢复。请访问 *http://dlc.sun.com/osol/docs/content/ZFSADMIN/gavvx.html*，以获取更多关于 ZFS 和执行备份和恢复的信息。

备份方法的比较

InnoDB Hot Backup、mysqldump 及第三方备份方法的区别体现在几个重要的方面，还有很多细微的差别。这里比较了各种备份方法，主要包括以下几个方面：它们是否允许热备份、它们的成本、备份速度、恢复速度、备份类型（逻辑或物理）、平台限制（操作系统）和支持的存储引擎。表 12-4 列举了本章提到的各备份方法及其比较项。

表12-4：备份方法的不同

	InnoDB 热备份	mysqldump	物理复制	XtraBackup	LVM/ZFS 快照
是否是热备份	是（仅限 InnoDB）	是	否	是（仅限 InnoDB 和 XtraDB）	是（需要刷新和锁定表）
成本	付费	免费	免费	免费	免费
备份速度	中等	慢	快	中等	快
恢复速度	快	慢	快	快	快
类型	物理	逻辑	物理	物理	物理

续表

	InnoDB 热备份	mysqldump	物理复制	XtraBackup	LVM/ZFS 快照
438 OS	所有的	所有的	所有的	所有的	仅 LVM 支持
引擎	InnoDB, MyISAM	所有的	所有的	InnoDB, XtraDB,MyISAM	所有的



Windows 上不完全支持 InnoDB Hot Backup。Perl 脚本在有些 Windows 配置中不能执行。详细信息请参照在线文档。

表 12-4 有助于规划数据恢复流程，帮助你找到满足需求的最佳工具。例如，如果需要在 InnoDB 数据库上进行热备份，并且成本不是问题，那么 InnoDB Hot Backup 将是你的最佳选择。另外，如果速度是主要考虑因素，要求备份所有数据库（所有存储引擎），并且操作系统为 Linux，那么 LVM 是个很好的选择。

备份和MySQL复制

使用 MySQL 复制进行备份有两种方法。在前面的章节已经介绍了 MySQL 复制及其在向外扩展和高可用性上的用处。本章介绍 MySQL 复制与备份有关的两个常见用途，包括使用复制创建数据备份副本，以及使用前面创建的备份进行 PITR（即时恢复）。

- 备份和恢复
 使用额外的服务器创建备份是很常见的，这样可以在不影响 Master 的情况下创建备份，因为你可以使备份服务器离线，然后执行任何操作。
- PITR
 即使定期创建备份，仍可能需要将服务器恢复到某个精确的时间点。通过合理地管理备份，确实可以将服务器恢复到指定的秒时间点。这对于恢复人为错误（例如敲错命令或输入不正确的数据），或者还原不必要的更改非常有用。各种情况都可能发生，它们都需要有正确的备份。

使用复制进行备份和恢复

一般来说，备份的缺陷在于只能在特定的时间创建（为了不影响其他操作，经常在夜间执行）。如果有个问题需要将 Master 恢复到备份创建后的 Master 某个时间点，你就惨了
439 吗？不是的！如果将备份与二进制日志结合使用，那么解决这个问题不仅仅是可能的，而且很容易。

二进制日志记录了在数据库运行时对该数据库所做的所有更改，所以，通过恢复正确的备份并将二进制日志回退到合适的时间点，就可以将服务器恢复到一个精确的时间点上。

当然，恢复过程中最重要的一步是恢复。所以在概述执行备份的流程之前，让我们先讨论如何执行恢复。

PITR

复制中的备份最常见的用途就是 PITR，即通过将系统恢复到最近正确的状态来使系统从错误（如数据丢失或硬件故障）中恢复，从而减少数据丢失。这么做的前提是至少进行了一次备份。

一旦修复了服务器以后，就可以还原最新备份映像，然后以二进制日志名及其位置为起点应用二进制日志。

下面描述了使用备份系统执行 PITR 的过程：

1. 将服务器还原到事件发生后的一个操作状态。
2. 找到你需要还原的数据库的最新备份。
3. 还原最新备份映像。
4. 使用 `mysqlbinlog` 工具应用二进制日志，将最新备份的开始位置（或开始时间）作为二进制日志的开始执行点。

这时你可能有个疑问："备份之后应该使用哪个二进制日志来执行 PITR 呢？"这取决于备份的方式。如果在运行备份前刷新了二进制日志，那么需要使用当前日志（最近打开的日志文件）的名称和位置。如果在运行备份前没有刷新二进制日志，则使用前一个日志的名称和位置。



更简单的情况下，PITR 在备份前总会刷新日志。这样，开始点就是文件的开头。

在错误被复制后还原

现在让我们看看备份如何帮助你恢复复制拓扑中的无意更改。假设某个用户做了一个灾难性（但却有效的）更改，并且这个更改被复制到了所有的 Slave 上。这时复制帮不上忙，但是备份系统可以拯救你。

下面的步骤用来恢复复制拓扑中数据的无意更改：

1. 删除 Master 上的数据库。

440

2. 停止复制。

3. 还原 Master 上的事件发生之前的最新备份。

4. 记录 Master 当前二进制日志位置。

5. 还原 Slave 上的事件发生之前的最新备份。

6. 在 Master 上执行 PITR，执行流程如上一节所示。

7. 从记录位置重启复制，并运行 Slave 同步。

总之，一个好的备份策略不仅仅是防止数据丢失的必要保护手段，同时也是一个重要的复制工具。

恢复示例

现在让我们看一个具体示例。假设你每天凌晨 2:00 定期创建备份，并将备份映像存储在某个地方备用。在这个例子中，假设所有的二进制日志都是可用的，一个都没有删除。实际上会定期清除二进制日志，以减少磁盘的占有空间。后面我们再考虑如何处理这种情况。

你已经将数据库恢复到 2009-12-19 12:54:23 的状态，因为这个时刻热心的助手不经意间将经理最心爱的照片删除了，她在清理桌子的同时也清理了经理的电脑。

1. 找出 2009-12-19 12:54:23 之前的备份映像。

2. 实际上，选择哪个备份映像并没有什么区别，但是为了节省恢复的时间，应该选择最近的备份，即当天上午的备份映像。

3. 还原备份映像，创建一个数据库在 2009-12-19 02:00:00 时刻的精确副本。

 找出 2009-12-19 02:00:00 到 2009-12-19 12:54:23 之间的所有二进制日志文件。开始时间之前和结束时间之后的事件都不重要，但是这些日志文件必须涵盖这个时间段内的所有事件。

4. 使用 `mysqlbinlog` 工具回放二进制日志文件，指定开始时间为 2009-12-19 02:00:00 和结束时间为 2009-12-19 12:54:23。

现在可以告诉经理他最心爱的照片回来了。

要想自动执行这个操作，必须做一些记录，这些记录将告诉你备份时需要保存什么。下面了解一下恢复时需要哪些信息。

- 为备份映像标记开始时间和结束时间以正确使用它们。这一点很重要，有助于选择使用哪个备份。 441
- 还需要备份的二进制日志位置。这个信息是必需的，因为回放二进制日志文件的开始时间不够精确。
- 还需要了解每个二进制日志文件记录信息的时间范围。严格地说，这不是必需的，但是很有帮助，因为它可以使你不必处理恢复映像的所有二进制日志文件。但是MySQL服务器不是自动执行的，你需要自己处理。
- 你不能永远保留这些文件，所以需要给所有信息、备份映像和二进制日志排序，这样当需要释放一些磁盘空间时，可以很容易地将它们归档。

恢复映像

为了帮助你管理可管理块中的备份信息，引入恢复映像的概念。恢复映像只是一个虚拟容器而不是一个物理实体：它只包含执行恢复必需的所有信息。

图12-2显示了一个恢复映像序列及每个恢复映像所包含的内容。序列中最后一个恢复映像比较特殊，称为开放恢复映像（*open recovery image*）。该恢复映像仍然处于添加更改的状态，因此它没有结束时间。其他的恢复映像称为封闭恢复映像（*closed recovery image*），且有结束时间。

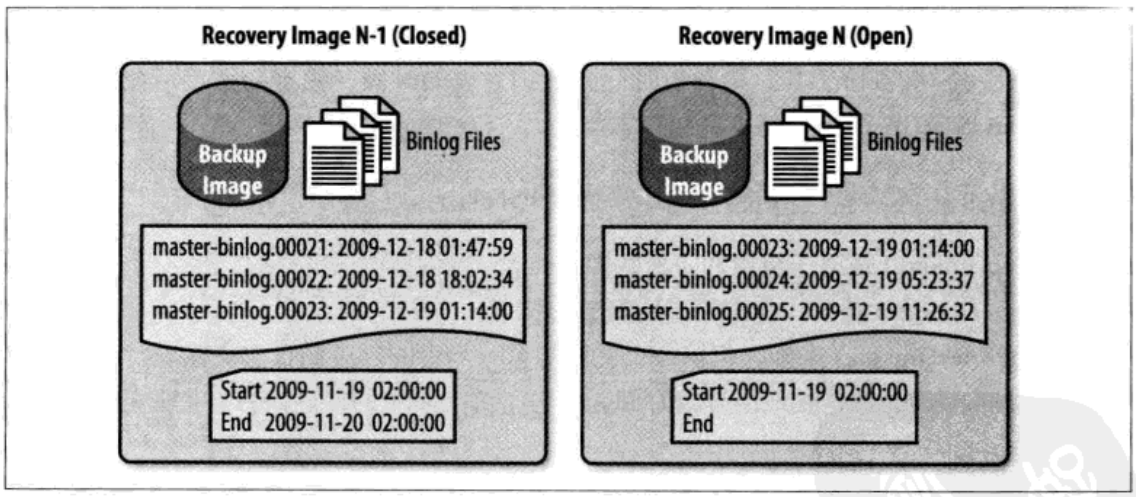


图12-2：恢复映像序列和内容

每个恢复映像都包含了执行恢复必需的信息： 442

- 备份映像
 备份映像在还原数据库时需要。

- 二进制日志文件集合
 二进制日志文件必须覆盖恢复映像的整个时间范围。
- 开始时间和可选的结束时间
 这些是恢复映像代表的开始和结束时间。开放恢复映像没有结束时间，这样不便于归档，所以我们认为它的结束时间就是当前时间。



二进制日志文件通常包含开始和结束时间范围之外的事件，但是恢复映像应该包含该时间范围内的所有事件。

- 每个二进制日志文件的名称和开始时间列表
 提取二进制日志文件需要每个文件的名称、开始时间和结束时间。可以使用`mysqlbinlog`提取二进制日志文件的开始时间。

备份过程

备份程序收集归档和恢复所需的所有信息和结构。假设已经有一个 Image1 到 $Image_{n-1}$ 的恢复映像序列，现在要创建第 n 个恢复映像。

1. 使用你最喜欢的方法创建备份，并记录备份映像的名称及其所代表的二进制日志位置。该二进制日志位置还包含二进制日志文件名。

 如果使用脱机备份工具，则备份时间就是锁定表的时间，锁定数据库后使用`SHOW MASTER STATUS`命令得到二进制日志位置。

2. 使用以下参数创建一个新的开放恢复映像 $Image_n$：

- Backup($Image_n$) 是第一步中的备份映像。
- Position($Image_n$) 是第一步中的二进制日志位置。
- BinlogFiles($Image_n$) 未知，但第一个文件是第一步中的二进制日志文件名。
- StartTime($Image_n$) 是映像的开始时间，来自于第一步中的二进制日志位置的事件。

443

3. 关闭 $Image_{n-1}$，请注意以下几点：

- BinlogFiles（$Image_{n-1}$）现在是从 Position($Image_{n-1}$) 到 Position($Image_n$) 的二进制日志文件。
- EndTime（$Image_{n-1}$）现在与 StartTime ($Image_n$) 相同。

Python中的PITR

为了统一管理各种备份方法，我们已经在例 12-1 中创建了 PhysicalBackup 类，这个类包含两个方法：

- `class PhysicalBackup.PhysicalBackup(image_name)`
 类的构造器，传入备份或恢复服务器所需的映像名称。
- `PhysicalBackup.backup_from(server)`
 该方法将创建服务器的备份，并存储该备份，该备份名称为构造器中传入的映像名。
- `PhysicalBackup.restore_on(server)`
 该方法将使用备份映像在服务器上备份和还原。

用类来表示备份方法，这样易于更换其他备份方法。

例12-1：表示物理备份方法的类

```
class BackupImage(object):
    "Class for representing a backup image"

    def __init__(self, backup_url):
        self.url = urlparse.urlparse(backup_url)

    def backup_server(self, server, db):
        "Backup databases from a server and add them to the backup image."
        pass

    def restore_server(self, server):
        "Restore the databases in an image on the server"
        pass

class PhysicalBackup(BackupImage):
    "A physical backup of a database"

    def backup_server(self, server, db="*"):
        datadir = server.fetch_config().get('datadir')
        if db == "*":
            db = [d for d in os.listdir(datadir)
                  if os.path.isdir(os.path.join(datadir, d))]
        server.sql("FLUSH TABLES WITH READ LOCK")
        position = replicant.fetch_master_pos(server)
        if server.host != "localhost":
            path = basename(self.url.path)
        else:
            path = self.url.path
        server.ssh(["tar", "zpscf", path, "-C", datadir] + db)
        if server.host != "localhost":
            subprocess.call(["scp", server.host + ":" + path, self.url.path])
        server.sql("UNLOCK TABLES")
```

444

```
        return position

def restore_server(self, server):
    if server.host == "localhost":
        path = self.url.path
    else:
        path = basename(self.url.path)

    datadir = server.fetch_config().get('datadir')

    try:
        server.stop()
        if server.host != "localhost":
            call(["scp", self.url.path, server.host + ":" + path])
        server.ssh(["tar", "zxf", path, "-C", datadir])
    finally:
        server.start()
```

接下来介绍恢复映像的表示，如例 12-2 所示。该恢复映像存储了 5 个字段：

- `RecoveryImage.backup_image`
 使用的备份映像。
- `RecoveryImage.start_time`
 恢复映像的开始时间。
- `RecoveryImage.start_position`
 备份映像表示的二进制日志位置。用该值代替开始时间作为回放二进制日志文件的开始点，这样比较准确。使用开始时间会出错，因为一秒内可能执行大量事务，它会选择标记为该时间的第一个事件，而真正的开始位置可能在其他地方。
- `RecoveryImage.binlog_files`
 恢复映像的二进制日志文件列表。
- `RecoveryImage.binlog_datetime`
 将二进制日志文件名映射到日期 / 时间的字典，即二进制日志文件中的第一个事件的日期 / 时间。

另外，恢复映像还必须包含以下方法：

- `RecoveryImage.contains(datetime)`
 判断恢复映像是否包含 `datetime`。由于二进制日志文件可能在半秒的时刻轮换，所以必须含有 `end_time`。

445

- `RecoveryImage.backup_from(server)`
 通过创建 `server` 的备份来创建一个新的开放恢复映像，并收集备份的相关信息。

- RecoveryImage.restore_to(server, datetime)
 还原服务器上的恢复映像，应用直到 datetime 之前（包括 datetime）发生的所有更改。这里假设 datetime 在恢复映像的时间范围内。如果 datetime 在恢复映像的开始时间之前，则不应用任何更改；如果 datetime 在恢复映像的结束时间之后，则应用所有的更改。

例12-2：一个恢复映像的实例

```
class RecoveryImage(object):
    def __init__(self, backup_method):
        self.backup_method = backup_method
        self.backup_position = None
        self.start_time = None
        self.end_time = None
        self.binlog_files = []
        self.binlog_datetime = {}

    def backup_from(self, server, datetime):
        self.backup_position = backup_method.backup_from(server)

    def restore_to(self, server):
        backup_method.restore_on(server)

    def contains(self, datetime):
        if self.end_time:
            return self.start_time <= datetime < self.end_time
        else:
            return self.start_time <= datetime
```

因为管理恢复映像需要处理多个映像，所以在例 12-3 中引入 RecoveryImageManager 类。除了构造方法外它还有两个方法：

- RecoveryImageManager.point_in_time_recovery(server, datetime)
 在 server 上执行 PITR，恢复到 datetime 时刻。
- RecoveryImageManager.point_in_time_backup(server)
 为 PITR 创建 server 的备份。

恢复映像管理器跟踪所有的恢复映像及使用的备份方法。这里假定所有的恢复映像使用相同的备份方法，但这不是必需的。

例12-3：RecoveryImageManager类

```
class RecoveryImageManager(object):
    def __init__(self, backup_method):
        self.__images = []
        self.__backup_method = backup_method
    def point_in_time_recovery(server, datetime):
```

```
    from itertools import takewhile
    from subprocess import Popen, PIPE

    for im in images:
        if im.contains(datetime):
            image = im
            break
    image.restore_on(server)

    def before(file):
        return image.binlog_datetime(file) < datetime

    files = takewhile(before, image.binlog_files)
    command = ["mysqlbinlog",
               "--start-position=%s" % (image.backup_position.pos),
               "--stop-datetime=%s" % (datetime)]
    mysqlbinlog_proc = Popen(mysqlbinlog_command + files, stdout=PIPE)

    mysql_command = ["mysql",
               "--host=%s" % (server.host),
               "--user=%s" % (server.sql_user.name),
               "--password=%s" % (server.sql_user.password)]
    mysql_proc = Popen(mysql_command, stdin=mysqlbinlog_proc.stdout)
    output = mysql_proc.communicate()[0]

def point_in_time_backup(self, server):
    new_image = RecoveryImage(self.__backup_method)
    new_image.backup_position = image.backup_from(server)
    new_image.start_time = event_datetime(new_image.backup_position)

    prev_image = self.__images[-1].binlog_files
    prev_image.binlog_files = binlog_range(prev_image.backup_position.file,
                                           new_image.backup_position.file)
    prev_image.end_time = new_image.start_time

    self.__images.append(new_image)
```

自动备份

自动备份是相当容易的。上一节演示了如何使用复制进行备份和恢复，本节将介绍备份和恢复的非复制方法的一般流程。

你可能会遇到的唯一问题是提供自动给备份映像文件命名的机制。解决办法很多，例12-4就给出了使用备份时间给文件命名的方法。将这个备份方法添加到Python库中以完善你的复制方法。这个库与前面章节中的库一样。

447

例12-4：备份脚本

```
#!/usr/bin/python
```

```
import MySQLdb, optparse

# --
# Parse arguments and read Configuration
# --
parser = optparse.OptionParser()
parser.add_option("-u", "--user", dest="user",
                  help="User to connect to server with")
parser.add_option("-p", "--password", dest="password",
                  help="Password to use when connecting to server")
parser.add_option("-d", "--database", dest="database",
                  help="Database to connect to")
(opts, args) = parser.parse_args()

if not opts.password or not opts.user or not opts.database:
   parser.error("You have to supply user, password, and database")

try:
   print "Connecting to server..."
   #
   # Connect to server
   #
   dbh = MySQLdb.connect(host="localhost", port=3306,
                         unix_socket="/tmp/mysql.sock",
                         user=opts.user, passwd=opts.password,
                         db=opts.database)

   #
   # Perform the restore
   #
   from datetime import datetime

   filename = datetime.time().strftime("backup_%Y-%m-%d_%H-%M-%S.bak")
   dbh.cursor().execute("[BACKUP COMMAND]%s'" % filename)
   print "\nBACKUP complete."

except MySQLdb.Error, (n, e):
   print 'CRITICAL: Connect failed with reason:', e
```



用你的备份解决方案替换可执行命令 [*BACKUP COMMAND*]。

如果不需要创建备份映像的名字，那么自动还原就更简单了。然而，依据你的安装、使用和配置，可能需要添加一些命令以保证与其他应用或用户之间的交互不受破坏。例 12-5 是一个可以加入 Python 库的常用还原方法，用来完善你的复制方法。

448 例12-5：还原脚本

```
#!/usr/bin/python

import MySQLdb, optparse

# --
# Parse arguments and read Configuration
# --
parser = optparse.OptionParser()
parser.add_option("-u", "--user", dest="user",
                  help="User to connect to server with")
parser.add_option("-p", "--password", dest="password",
                  help="Password to use when connecting to server")
parser.add_option("-d", "--database", dest="database",
                  help="Database to connect to")
(opts, args) = parser.parse_args()

if not opts.password or not opts.user or not opts.database:
   parser.error("You have to supply user, password, and database")

try:
   print "Connecting to server..."
   #
   # Connect to server
   #
   dbh = MySQLdb.connect(host="localhost", port=3306,
                         unix_socket="/tmp/mysql.sock",
                         user=opts.user, passwd=opts.password,
                         db=opts.database)

   #
   # Perform the restore
   #
   from datetime import datetime

   filename = datetime.time().strftime("backup_%Y-%m-%d_%H-%M-%S.bak")
   dbh.cursor().execute("[RESTORE COMMAND]%S'",
                        (database, filename))
   print "\nRestore complete."
except MySQLdb.Error, (n, e):
   print 'CRITICAL: Connect failed with reason:', e
```



用你的备份解决方案替换可执行命令 [*RESTORE COMMAND*]。

从例 12-5 可以看出，可以自动执行还原。然而，大多数人喜欢手动执行还原。如果在测试环境下从基线开始恢复，并且需要将数据库还原到某个可知状态，这时自动还原可能

比较有用。另外，自动还原还可以用于开发系统中为项目保留特定的环境。

小结

本章学习了IA，并研究了它对IT专业人士最有影响的几个方面。了解了灾难恢复规划的重要性，如何制订自己的灾难恢复计划，以及数据库系统为什么是灾难恢复不可分割的一部分。最后，研究了通过定期备份来保护MySQL数据的几种方法。

接下来的几章将讨论更高级的MySQL话题，包括MySQL企业版、云计算和MySQL集群。

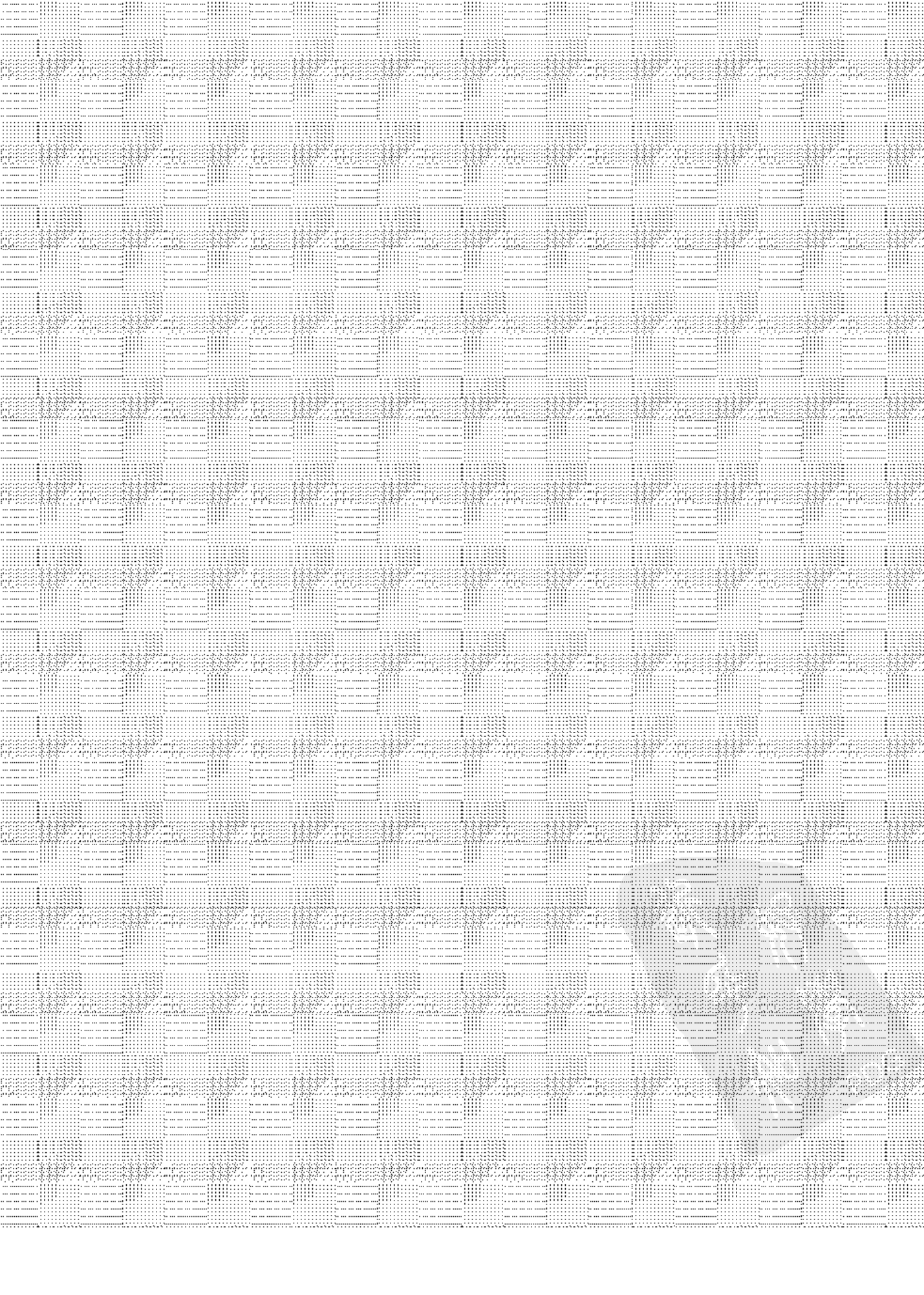
Joel看了一眼终端窗口，又敲了个命令查看他的数据库备份。他建立了一个循环备份脚本来备份所有的数据库，相信这个简单的操作没什么问题。他叹息道这仅仅只是他的恢复计划的开始，希望多写些脚本进行试验，然后安排他的第一个灾难计划会议。他已经想好了几个小组成员。突然，一阵敲门声打断了他的思路。

“订购媒体了吗，Joel？那个计划怎么样了？”Summerson先生问道。

Joel笑了笑说：“是的，我确定我可以备份整个数据库了，使用……”

“好的，很好，Joel。我不需要知道细节信息，只要远离审计员就好了，是吧？”

Joel笑着点点头，然后老板就去询问另一个员工的工作了。Joel怀疑他的老板是否真的了解要达到目标需要多大的工作量。他打开邮箱，开始写邮件请求分配更多的人手和资源。



第13章

MySQL企业版

Joel打开另一个终端窗口观察输出。他揉揉眼睛放松了下视神经。当他试图监控分散在三个地点所有的服务器时，屏幕上的数字陷入了混乱，过了很久他才拿到需要的报告。

当只有几台服务器的时候，他曾试图使用电子表格来存储数据。但现在已经超过30台服务器，工作也随之枯燥起来。他的朋友曾试图说服他购买企业级的监控工具套件，但老板认为这样无法立即给公司带来收入，所以拒绝了该项议案。

“嗨，Joel。”

Joel抬起头，看到他在客户支持部门的朋友Doug斜靠在他的门边，手里还拿着杯咖啡。“嗨，”Joel说。

“你看上去需要休息一下。”

“没时间啊。我需要写一份所有服务器的状态报告，但是不检查完所有服务器，我写不出来。”

“哇，这样干真费劲。”

“嗯，我知道有些企业套件能很容易地解决这些问题，但即使Summerson先生批准，我也不知道该买哪一种。“

“嗯，如果你能向他证明这一切需要多少时间……”Doug一边说，一边激动地摇着他的咖啡杯。

Joel 想了一会儿。“如果我能向他证明通过我辛苦努力做出来与用一个好工具做出来之间的区别……”

“好主意。现在来一杯怎么样？我请客。”

Joel 跟着 Doug 来到休息室，并与他聊起了最近参加的大卫马修乐队演奏的音乐会。

Joel 回到办公室后开始了解 Oracle 公司提供的 MySQL 企业版。

452 监控一组服务器的工作量很大。有很多可以妥善监管服务器的工具，虽然只要你了解了它们是做什么用的，就能很容易地用起来，但是仍需要做点工作才能运行它们。脚本和一些有创意的 Web 应用有助于管理各种工具的运行及数据的收集，但是随着被监控服务器的数量增加，搜集和分析所有数据将变得十分耗时。

目前为止，在本书中讨论过的监控技术还不能管理很多服务器。事实上，在一个拥有很多服务器的组织机构中，由多个全职技术人员进行手动监控和报告。



本章主要讨论监控，并涉及部分管理。一般来说，如果使用管理工具自动执行命令，例如运行脚本来填充表、执行表维护等，可以大大减轻管理员的负担。

幸运的是，这个问题不是才出现的，也不是不能解决的。事实上，有几种企业监控套件可以很容易地监控大量服务器。

其中最容易被忽略的监控 MySQL 工具是 MySQL 企业监控器（MEM），它已经附带在 MySQL 企业版中。MEM 工具可以大大改善监控和预防性维护，并大大减少诊断时间和停机时间。虽然这是一个付费工具，但是一个维护良好的数据中心节约下来的钱要远远超过购买该工具的开销。

本章介绍 MySQL 企业版套件的工具，并展示这些工具是如何在保持 MySQL 服务器的性能和可用性最高的情况下，有效地节约时间的，还将介绍在复杂复制拓扑结构上运行监控工具的实例。

MySQL企业版入门

MySQL 企业版是在 2006 年启动的，包括企业 MySQL 服务器发行版、一套监控工具及产品支持服务。这个新包是为使用 MySQL 管理数据的客户设计的。在早期，MySQL 认识到组织机构对稳定性和可靠性的需求，MySQL 企业版刚好满足这个需求。



如果你尚不准备购买 MySQL 企业版，或者想试用一段时间后再考虑买不买，可以获得一个为期 30 天的试用版。请在 *http://mysql.com/trials/* 上申请试用版。

MySQL 企业版添加了基于 Web 的 MEM 应用，还有一个单独的 MySQL 服务器实例用来存储 metric，这些 metric 是通过 MySQL 服务器上称为代理（*agent*）的其他应用程序收集的。MEM 将这些 metric 整合到报告中，然后使用称为 *advisor* 的启发式信息改善报告，这些 *advisor* 有助于强化基于 MySQL 研究和专家知识的最佳实践。报告显示在被称为 *dashboard* 的网页上。我们将在本章的后面讨论如何安装和使用这些组件。 453

下面描述了 MySQL 企业版订阅服务的可用选项，并概述了其安装流程。后面的章节更详细地描述了它的特性和益处。

订阅级别

MySQL 企业版套件有 4 种收费版本，即基本版、白银版、黄金版和白金版。基础版是最便宜的，它提供的功能也最少。其他三个版本依次提供一个比一个更全面的功能支持。因此，你可以从中选择既能够满足预算又能提高服务器的可用性的版本。

- 基本版
 包括 MySQL 服务器专业版，提供 MySQL 服务器的基础功能，包括软件更新和两次事故支持。支持两个工作日内的往返和响应时间。还提供对 MySQL 知识库的访问，以研究常见问题及其解决方案。但是该版本不包含监控工具。基本版最适合如下组织机构：需要产品质量保证但是可能不需要高级监控或对问题的立即响应。
- 白银版
 包括基本版的功能，包含有限的管理和升级 advisor 的监控工具。还包含电话技术支持功能，并提供无限制的事故报告，且往返时间是四小时。白银版比较适合以下企业：依靠 MySQL 进行数据库管理，并希望确保进行正确配置安装的企业。
- 黄金版
 加入了 MySQL 服务器高级特性（包括分区功能），包含白银版的所有功能，以及监控复制和内存的 advisor 功能。还为监控提供了 MySQL 查询分析工具。另外还提供复制和分区方面的咨询支持。事故报告的响应时间是两小时，还可选择紧急响应选项。黄金版比较适合需要使用 MySQL 复制的企业等组织机构。
- 白金版
 包括黄金版的所有功能，以及所有的 advisor 功能，提供全方位 24×7 的电话支持，提供一小时事故的响应时间，紧急事故则为 30 分钟。该版甚至提供 MySQL 服务器的自定义安装功能。白金版适合最重要的数据管理需要最高级别的技术支持的组织 454

机构。

MySQL 提供了这么多选项，你可以随着业务的增长加入更多的支持功能。从 *http://mysql.com/products/enterprise/features.html* 可以找到各个版本的价格及其功能。

安装概述

购买 MySQL 企业版后，就可以获得登录 MySQL 企业门户网站的产品密钥和登录资格证书。激活 MySQL 企业版工具必须要连接到这个门户网站。这个门户网站提供一站式的服务：下载更新，检查升级、新闻和版本信息，以及访问知识库。

该门户网站名为 MySQL 企业用户中心，地址为 *https://enterprise.mysql.com/*。



如果没有网络，需要脱机安装 MySQL 企业版，可以从这个门户网站下载产品密钥文件，然后用它进行安装。还需要绑定指导性 *.jar* 文件，这些文件都可以从门户网站下载。

首先到这个网站下载特定平台下的企业工具安装文件、MySQL 服务器发行版、入门手册和企业文档。一定要阅读入门手册，因为它提供了在 Mac OS X、Linux 和 Windows 平台上安装和配置 MySQL 企业工具的具体说明。

本书不提供安装说明的完整详细信息，不过我们简要描述了安装和配置 MySQL 企业套件的必要步骤。安装过程如下（至少包含以下步骤）：

1. 安装服务管理组件，包括 metric 库（需要独立安装），该组件与操作系统有关。例如，Mac OS X 系统上安装的文件名为 *mysqlmonitor-2.1.0.1096-osx-installer*。

455

2. 激活并启动 Enterprise Dashboard。这一步需要输入产品密钥，你可以导入密钥文件或者登录 MySQL 企业门户网站。

3. 在 MySQL 服务器上安装监控代理。监控代理也与操作系统有关，例如，Linux 系统上安装的代理名为 *mysqlmonitoragent-2.1.0.1093-linux-glibc2.3-x86-32bit-installer.bin*。

4. 配置 Dashboard，显示你的环境信息。

尽管可以在任何服务器上安装监控代理来监控其他服务器，但最好将监控代理安装在需要监控的服务器上，这样代理可以将操作系统统计信息、性能数据和配置参数发送到 Dashboard 上。如果在其他服务器上安装代理，就没有关于 MySQL 服务器系统本身的 Dashboard 报告了。



可以将代理配置为：为多个服务器报告统计信息，或将报告发送到多个服务器上。前面一种配置允许单个代理监控单个系统上的多个 MySQL 服务器实例；后面一种配置允许多个 Dashboard 从 MEM 报告数据。这两种情况在 MEM 参考手册上都有描述。

MEM 包的安装包括自带的 Web 服务器，以及 Dashboard 和 metrics 库所在的系统上的 MySQL 实例，且该系统是所有监控代理的目标。这个安装过程很简单，不需要任何网络管理的专业技术。

安装监控包时，必须指定账户名和服务器地址（比如 IP 地址）。安装监控代理时需要这些信息。入门手册中有相关说明，要特别注意其中的例子。可以打印入门手册，并将安装过程中提供的选项信息记下来供以后参考，这样或许有帮助。

监控代理的安装也很简单。如果监控服务器开始运行，且产品密钥仍然有效，就可以为网络中的每个 MySQL 服务器都安装一个代理。有些系统可能需要手动启动代理，这一点在手册中有详细介绍。

只要安装和启动了一个代理，就可以到 Enterprise Dashboard 中配置代理，使其满足要求。

MySQL企业组件

456

MySQL 企业版由 MySQL 服务器软件、MEM 工具集和产品技术支持组成。

MySQL企业服务器

MySQL 企业版包含两个版本的 MySQL 服务器：专业版是具有最稳定功能集的发行版本；高级版在专业版的基础上还添加了一些试验性的功能，例如表和索引的水平分割，提高了大型数据库的性能。

MEM

MEM 是服务器连续监控和警告的核心。MySQL 网站上是这样形容它的："它就像你身边的虚拟的 DBA 助手，可以向你推荐减少安全漏洞、提高复制性能、优化性能等的最佳解决方案。"如果不考虑市场营销，MEM 可以提供满足日益增长的业务数据中心需求的专业工具。

MEM 包括以下几个主要功能：

- 监控所有服务器的健康状态，并以单一的网页形式显示。

- 600 多个关于 MySQL 服务器及其运行的操作系统的指标。
- 监控性能、复制、模式和安全的能力。
- 通过简单的热图显示服务器的实时健康状态。
- 超过 metric 阈值告警。
- 实施 MySQL 创建者的最佳实践规则集。

MEM 由运行在内部网络上的分布式 Web 应用程序组成。每个向 Web 服务器组件发送 metric 的 MySQL 服务器都会安装一个监控代理，即 Enterprise Dashboard。这里有所有描述服务器状态的统计信息和图形。MEM 还包含实施最佳实践的指导顾问，确保服务器的配置合理，并能高性能运行。

Enterprise Dashboard

MySQL 企业版工具的图形客户端是 Enterprise Dashboard，它是运行在监控服务器上的 Web 应用程序。Enterprise Dashboard 提供了一个监控所有服务器的可用性、安全和性能数据的方法，它可以有一个或多个。这里可以查看每个服务器的相关健康状况、性能和内存使用率的状态图，以及每个服务器操作系统的重要统计信息。

457 Enterprise Dashboard 以简单易读的格式显示监控和警告信息。图 13-1 是一个 MEM 的 Enterprise Dashboard 简单实例。

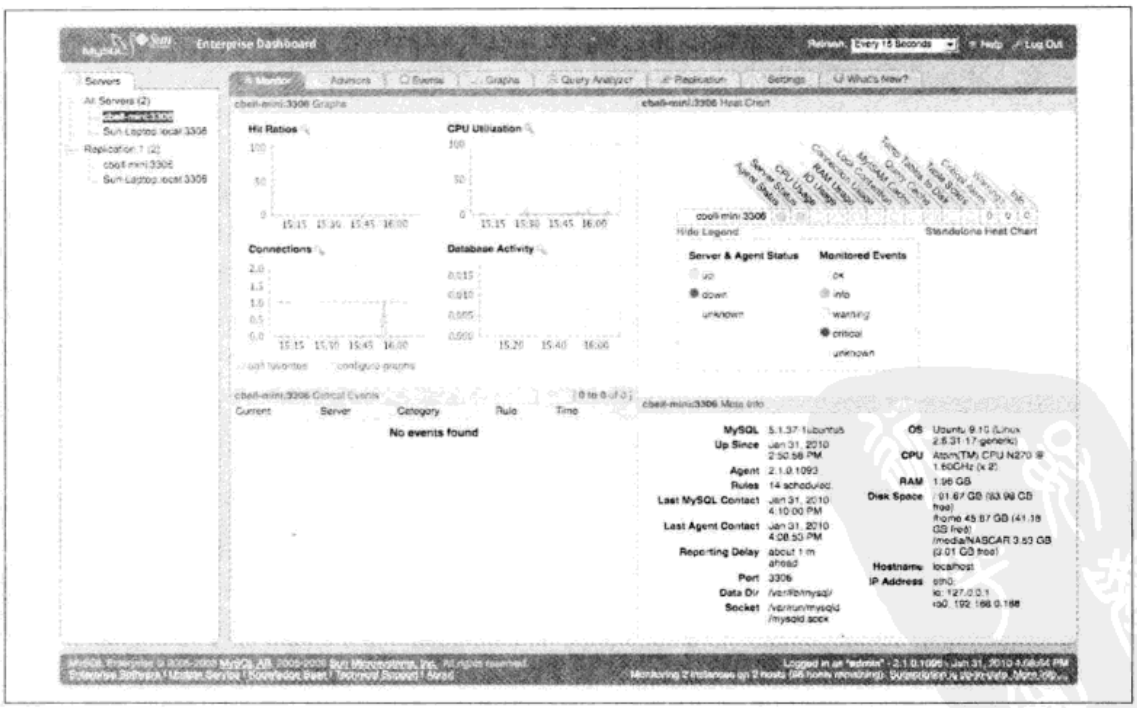


图13-1：MySQL Enterprise Dashboard

可见，Enterprise Dashboard 在一个窗口上提供了所有的重要信息，包括 advisors、事件、附加性能图、查询分析器、复制，以及配置设置等标签。What's New 标签包含了 MySQL 工具和企业版相关新闻和事件的链接。

监控代理

监控代理是个特殊的轻量级应用，负责收集 MySQL 服务器的相关信息，包括主操作系统的统计信息等。因此，监控代理是监控工具的主要组件，想在每个需要监控的服务器上安装监控代理，就要求这个代理不仅仅是轻量级的，而且几乎是透明的，即监控代理不能带来明显的性能下降。

Advisors

MySQL 企业版工具有一个与典型企业监控方案不同的功能，即监控系统特定区域的性能和配置，并在服务器偏离 MySQL 设计者定义的最佳状态时发出警告。也就是说可以在系统配置、安全或性能出了问题时立即得到反馈。该功能称为 advisor，可以针对不同 458 方面进行监控和报告，包括：

- 管理
 监控一般的数据库管理和性能。
- 升级
 监控升级情况并针对潜在的版本问题发出警告。还提供修复关于升级问题的特定错误的策略。
- 性能
 将当前 MySQL 服务器性能与 MySQL 性能的最佳实践规则对比，找出差距。
- 模式
 识别对数据库和模式对象的更改。可以监控这些更改，并在多余的和意外更改发生时发出警告。
- 内存使用情况
 识别内存使用情况的变化，并在性能降低时发出警告。
- 安全
 识别并警告潜在的安全漏洞。
- 复制
 识别与配置、健康、同步（延迟）和性能问题相关的复制情况
- 自定义
 还可以创建自定义 advisors 以支持你的最佳实践或满足特定需求。

每个 advisor 使用最佳实践规则集全面涵盖了服务器某个方面的信息。advisor 有助于

识别服务器什么地方需要注意，并给出了如何改进或修正当前状态的建议。如果这组advisor集还不够全面，可以根据需求创建自己的advisor。

查询分析器Query Analyzer（黄金版和白金版）

复杂的数据库和应用程序可能需要执行复杂查询。如果使用SQL表达式，通常写出来的查询语句效率都不高。而且，写得很差的查询往往会降低性能。专业的数据库管理员认识到这一点，他们通常在进行数据库性能诊断时首先检查查询语句。

一般可以通过查看慢查询的日志或进程列表（如执行`SHOW PROCESSLIST`命令）来发现低性能的查询。一旦发现了需要注意的查询，就可以使用`EXPLAIN`命令查看MySQL是
459 如何执行该查询的。虽然这个过程众所周知，而且一度被数据库管理员使用，但这是一个很费力的任务，且不容易写成脚本。随着数据库的复杂性的增加，低性能查询的诊断也越来越费力。

你可以使用这种方法检查用户查询，但是如何审查包含在应用程序代码中SQL语句的性能呢？这种情况是最难诊断和修复的，因为它需要更改应用程序。假定可以更改应用程序，如何建议开发者改进查询？

不幸的是，很少有人怀疑应用程序代码中的SQL语句是影响性能的根源。当遇到性能问题时，DBA和开发人员都急于责怪服务器或系统，而不是嵌在应用程序中的SQL查询。更糟糕的是，MySQL系统不支持强大的性能metric集，也不支持麻烦查询的发现。

如果能找出服务器上所有长时间运行的查询，并检查最慢的查询，岂不是更好？企业监控工具的查询分析器可以做到这一点。

可以在Enterprise Dashboard上启动查询分析器。查询分析器的安装和配置需要一点工作，所以一定要参考入门手册了解详细信息。

查询分析器实时显示查询的性能统计信息。它在一个地方显示来自所有服务器的所有查询的信息，所以不需要逐个服务器地寻找低性能的查询。这个列表还维护了所有查询的历史，这样就不必担心日志的存储空间问题。

每个查询都有两种不同视图：一个是规范视图（没有数字数据），以图像的方式显示查询；而另一个视图显示查询的具体时间和访问数据。最重要的是，advisor甚至可以提醒你查询什么时候在哪个服务器上执行。

查询分析器可以发现性能不好的查询，然后通过检查可疑查询的执行计划来确定解决方案。显然，光这个功能就可以节省大量时间，特别是在以下情况下：开发或优化部署应用程序，或者需要优化查询以提高性能。

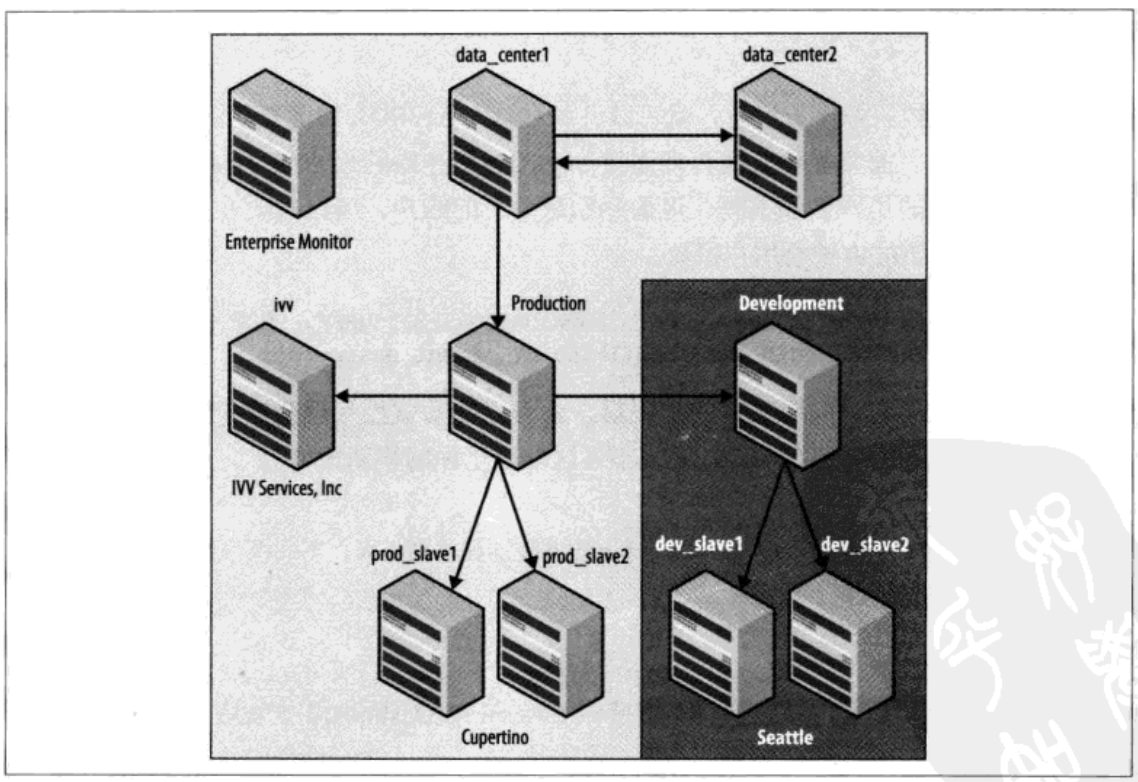
MySQL产品支持

MySQL 企业版还包括专业技术人员的支持，技术支持人员可以帮助你开发、部署和管理你的 MySQL 服务器。技术支持包括问题解决、咨询、访问常见解决方案的在线知识库以及（白金版才有）技术客户经理，他是帮助你解决所有 MySQL 问题的联络人。

MySQL企业版的使用

现在已经介绍了 MySQL 企业版的功能及其组件，接下来通过实例看看这些工具是如何使组织机构受益的。在这个例子中，假设基于 Web 的公司已经具备信息基础设施，这并不罕见。这个例子是行业中复制模型的典型代表，包括一个复杂的多 Master 复制配置，并支持多个数据库系统之间的复制（只部分复制某些数据库）。

如图 13-2 所示，每个数据中心由两个 Master 组成，提供高可用性和负载均衡。这两个数据中心托管公司的数据库，每个数据中心存储不同产品线的数据。Slave 连接到主数据中心，以获取内部产品信息和日常操作。这个 Slave 也有自己的部门 Slave，包含提供独立的检验和确认（Independent Verification and Validation，IVV）服务的第三方 Slave。连接到相同 Master 的还有另外一个 Slave，该 Slave 主要供开发部门使用，负责建立和改善产品线。



13-2：信息基础设施示例

数据中心中的每个 Slave 都可以（而且一般是这样做的）包含其他未复制的数据库。例如，产品服务器通常承载人力资源数据库，而这些数据不会被复制到大部分 Slave 上（例如不会被复制到开发中心）。同样地，第三方服务器承载它自己的结果数据，而开发服务器拥有不同的开发状态下的产品线数据库的各种版本。

安装

安装 MySQL 企业版需要建立数据库服务器来运行最新的专业版或高级版。可以使用已经安装的 MySQL 服务器，但是为了得到最佳回报，应该使用 MySQL 企业版提供的安装版本。一旦配置好了数据库服务器，并能正常运行，就可以开始安装 Enterprise Monitor 和监控代理了。

首先将 MEM 安装在网络上的一台服务器上，并连接所有需要监控的服务器（通常推荐使用 MEM 监控你的 MySQL 服务器）。在安装过程中，一定要记下该服务器的主机名和 IP 地址，以及代理访问所需的用户名和密码。安装过程很简单，在 MySQL 企业版网站上有详细介绍。

461



在 MySQL 企业版安装过程需要几个用户账号，除了企业版的认购账号外，还需要使用 MEM 管理员账号、代理访问 MEM 服务器的账号和代理访问 MySQL 服务器上的监控代理的账号。如果把这些账号搞混淆了，将会导致安装失败。

一旦 MEM 在监控服务器上运行，就可以开始在每个 MySQL 服务器上安装监控代理了。安装监控代理时，需要提供监控代理连接 MySQL 服务器的用户账号和密码。所有服务器最好使用相同的用户名和密码，但是它们是单个的账户，所以需要在每个服务器上创建账号，并用如下的方式给用户授权：

```
GRANT SELECT, REPLICATION CLIENT, SHOW DATABASES, SUPER, PROCESS ON *.*
TO 'agent_user'@'localhost' IDENTIFIED BY 'agent_password';
```

462

创建了账号并授予它访问服务器的权限后，就可以启动监控代理观察 MEM 了。这个服务器应该在几分钟内显示在 MEM 上，不过这取决于你的刷新设置。



MEM 和监控代理在各个平台上的安装过程差别不大。不同操作系统上具体如何安装请参看相关文档。

在每个服务器上重复上述安装过程，然后在 Enterprise Dashboard 上观察显示的结果。图 13-3 显示了包含所有监控代理报告信息的示例的 Enterprise Dashboard。

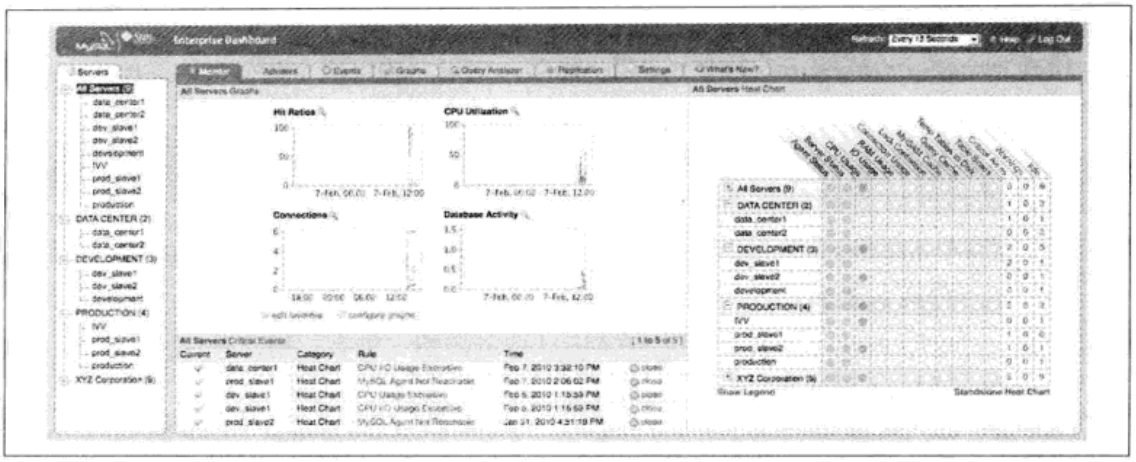


图13-3：Enterprise Dashboard

图 13-3 右边的热图中显示了每个服务器及其代理的状态，图中绿色点表明状态良好。图中有四个图表，分别描述了所有服务器的查询缓存命中率、CPU 使用率、连接信息和数据库活动的综合信息。这些图下面是 advisor 报告的默认重要事件列表，其中有关于 CPU、I/O 和内存的警告。

可以推断，此页给出了服务器的所有相关信息，这样你就可以一眼看出你的信息基础设施的整体健康状况。显然，没有比这更容易的监控了。

修复监控代理问题

虽然监控代理的安装过程非常简单，但是有时也会出错。小心地提供正确的信息，会一切顺利。

以下是监控代理向 MEM 提交报告的过程中，诊断和纠正问题的基本流程：

463

- 如果监控代理开始运行了，但服务器没有被显示在 Enterprise Dashboard 上，或者如果在热图中代理或服务状态显示有错误（即红色点），那么请检查 *mysql-monitor-agent.log* 文件，其中含有大量有助于解决大部分问题的信息。日志文件位于以下位置：
 - *Mac OS X*
 /Applications/mysql/enterprise/agent/
 - *Linux/Unix*
 /opt/mysql/enterprise/agent on Linux
 - *Windows*
 C:\Program Files\MySQL\Enterprise\Agent

- 检查访问 MySQL 服务器上的监控代理的用户账号和权限。
- 检查 *agent-instance.ini* 文件中的用户账号和密码，确保这个账号与你监控 MySQL 服务器使用的账号一致。
- 验证本地 MySQL 服务器的端口和主机名。确保与 *agent-instance.ini* 文件中的信息相同。
- 检查 *agent-instance.ini* 文件中的服务器端口。确保可以使用该文件中的端口、用户名和密码登录本地 MySQL 服务器。
- 检查 *mysql-monitoragent.ini* 文件中 MEM 服务器的主机名、用户名和密码。确保可以 ping 通 MEM 服务器。

如果查询分析器无法运行，还要检查 *mysql-monitoragent.ini* 文件中的代理端口，确保使用该文件中的指定信息 MySQL 客户端可以连接到代理。

监控

MySQL 企业版有几种简化监控的方法来管理复杂的信息基础设施，包括：

- 热图
- 警告信息
- 综合服务器图
- 服务器详细信息
- 复制详细信息
- 顾问（Advisor）

下面几节将详细介绍以上几个方面。

464



使用设置页中的管理服务器选项来重命名服务器。这样就可以在 Enterprise Dashboard 上使用更有意义的名称，而不改变真正的服务器主机名。

还可以创建组，用来合并相关的服务器。这样在不同的控制面板上显示组时，可以折叠组或者根据需要改变其显示。

热图

如前所示，Enterprise Dashboard 监控页面右侧的热图提供了服务器健康状况的概览。该图（你可以打开或关闭）以不同颜色表示系统运行的状态，绿色表示联机和全面运作中，红色表示脱机或无通信。显然，从中你可以很快发现那些需要进一步检查的区域。图 13-4 显示了一个信息基础设施的热图图例。

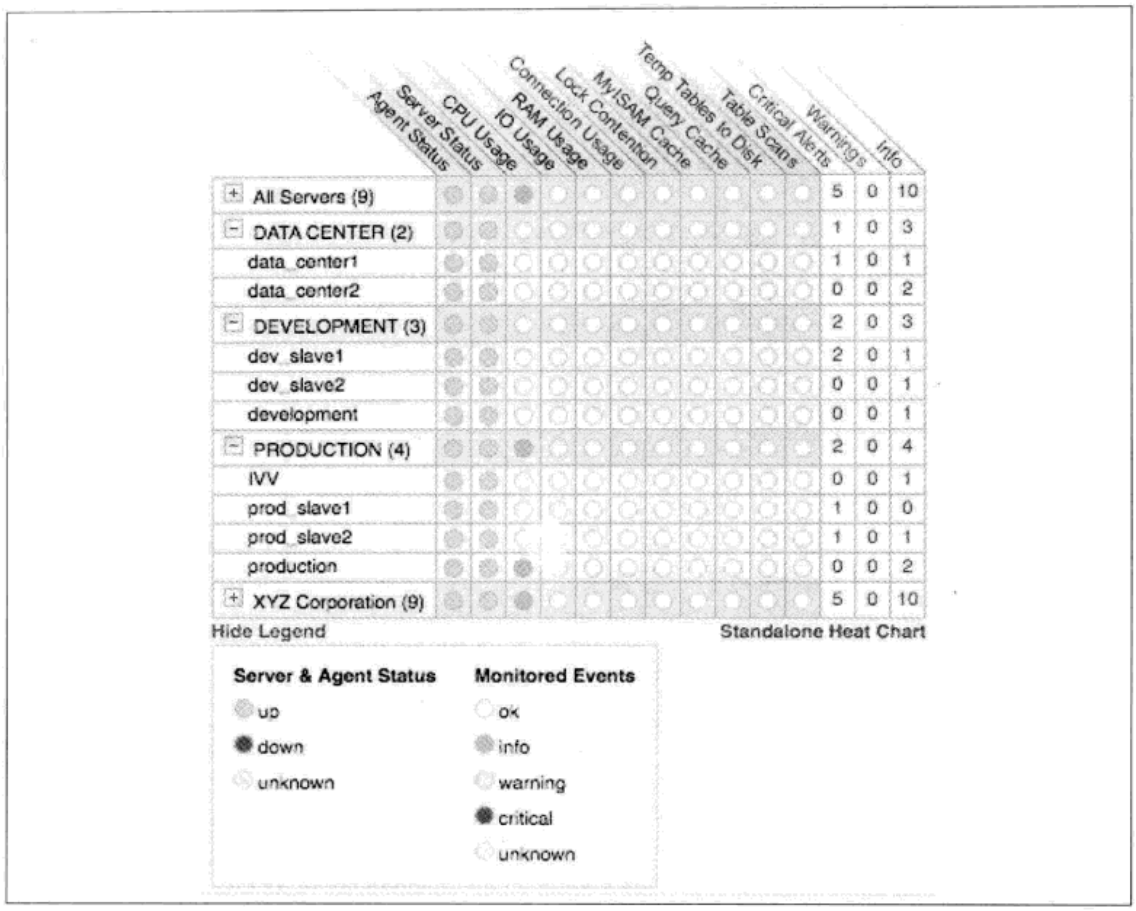


图13-4：散热图

请注意，热图顶部列出了关键监控区域的类别，这些区域涵盖了第 8 ~ 10 章中讨论的监控相关的重要方面（如 CPU、内存等）。与手动监控方法不同，热图表示了相对健康度量，可以快速获取状态信息。

除了一般监控区域外，还有一些 MySQL 特定的监控区域，如锁竞争、MyISAM 缓存利用率、查询缓存使用率及表扫描的数量。当然 MySQL 企业版可以监控报告更多信息，但是主要监控以上几个区域。

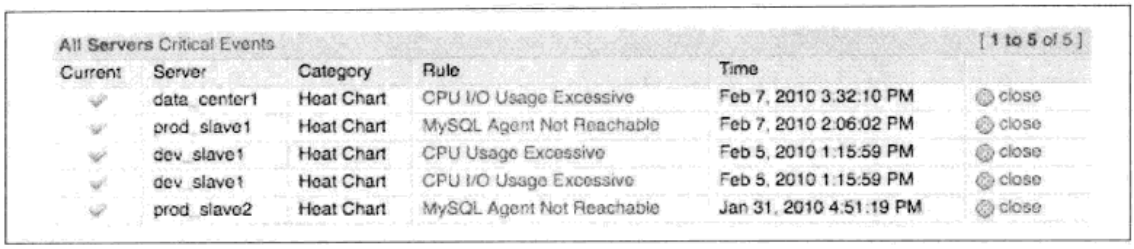
类别的右侧列出了最近重要事件、警告和消息的计数信息。



热图中的值是定期更新的，所以最近行为及问题的解决将会导致计数变化。

警告信息

热图的最大好处在于你可以单击任何一个状态点或数据条以获取更多的信息。例如，如果服务器遇到 I/O 问题，单击 I/O 使用率点，将看到系统的所有警告列表，如图 13-5 所示。然后单击最新的警告，得到如图 13-6 所示的详细报告。

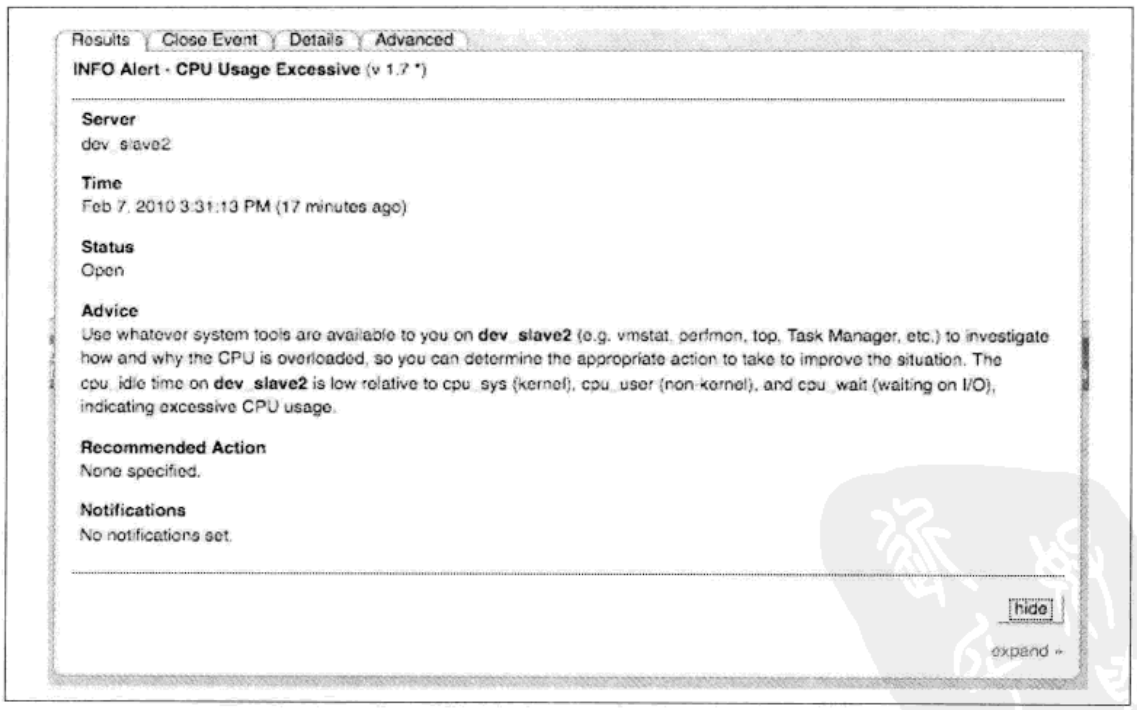


All Servers Critical Events [1 to 5 of 5]

Current	Server	Category	Rule	Time	
✓	data_center1	Heat Chart	CPU I/O Usage Excessive	Feb 7, 2010 3:32:10 PM	close
✓	prod_slave1	Heat Chart	MySQL Agent Not Reachable	Feb 7, 2010 2:06:02 PM	close
✓	dev_slave1	Heat Chart	CPU Usage Excessive	Feb 5, 2010 1:15:59 PM	close
✓	dev_slave1	Heat Chart	CPU I/O Usage Excessive	Feb 5, 2010 1:15:59 PM	close
✓	prod_slave2	Heat Chart	MySQL Agent Not Reachable	Jan 31, 2010 4:51:19 PM	close

图13-5：警告列表实例

465 这份报告显示了发生警告的服务器、警告发生的时间，以及遵循既定的最佳实践的建议。顶部有一些选项卡：单击 Close Event 选项卡可以清除显示面板上的警告信息（关闭那些已经被修复的或被接受的事件的警告），单击 Details 选项卡还可以查看更多的信息（如某个问题的扩展描述），单击 Advanced 选项卡可以查看这些警告是如何被触发的。



Results | Close Event | Details | Advanced

INFO Alert - CPU Usage Excessive (v 1.7 *)

Server
dev_slave2

Time
Feb 7, 2010 3:31:13 PM (17 minutes ago)

Status
Open

Advice
Use whatever system tools are available to you on **dev_slave2** (e.g. vmstat, perfmon, top, Task Manager, etc.) to investigate how and why the CPU is overloaded, so you can determine the appropriate action to take to improve the situation. The cpu_idle time on **dev_slave2** is low relative to cpu_sys (kernel), cpu_user (non-kernel), and cpu_wait (waiting on I/O), indicating excessive CPU usage.

Recommended Action
None specified.

Notifications
No notifications set.

hide

expand »

图13-6：警告报告样例

466 警告报告使 MySQL 企业版的监控功能脱颖而出，所以它被称为“虚拟的数据库管理助

手”。警告使监控更容易，它可以捕获组织机构中的服务器问题，然后在同一个地方显示这些问题。这样节约了时间，也避免了高成本诊断或行为监控的乏味，并提供了如何快速解决问题的提示。

服务器综合图

Enterprise Dashboard 的中间显示了一个综合图，以不同颜色的线条代表每个被监控的服务器（如图 13-7 所示）。在默认设置下，这些图很小，但是可以改变其大小和报告的时间尺度。

使用这些图可以得到组织机构的另一个系统健康图。即使在默认的小图上，也很容易发现异常，可以单击服务器综合图查看每个事件的详细信息。

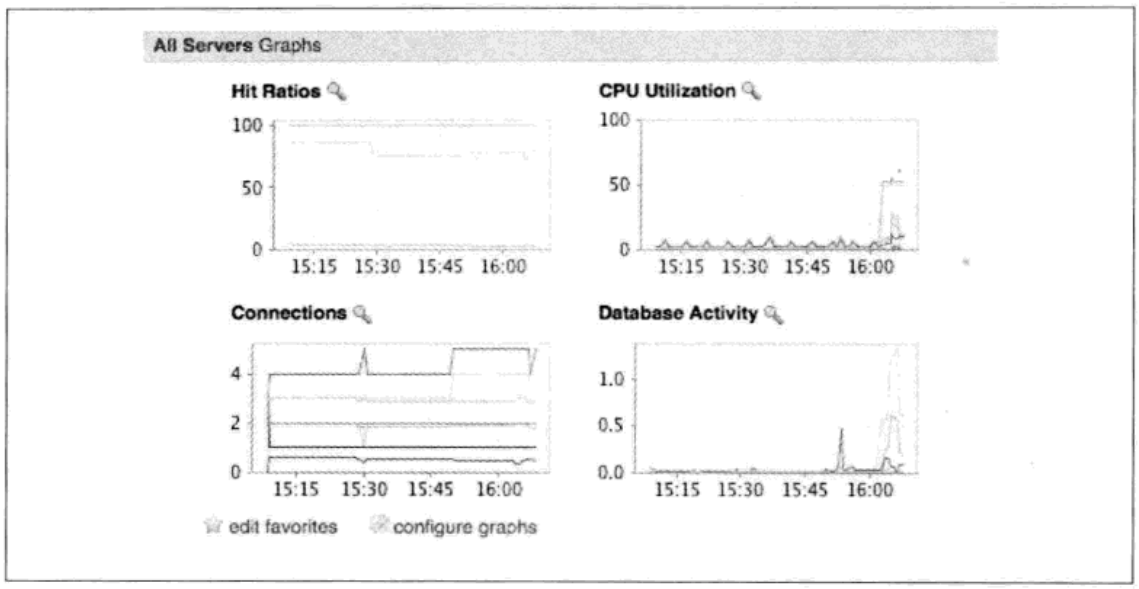


图13-7：综合服务器图的样例

服务器详情

Enterprise Dashboard 的另一个很棒的功能是：可以单击服务器列表中某个特定的服务器，查看该服务器系统的更多详情。服务器详情报告中显示了以下信息：MySQL 服务器的版本、MySQl 服务器最后一次启动的时间（运行时间）、数据存放的位置、主机操作系统、CPU、内存大小、磁盘空间大小和网络信息。

可以通过这些信息列一个清单（确定网络上有什么硬盘），还可以快速了解 MySQL 服务器运行在什么操作系统上，从而为解决问题提供线索。例如，要远程登录某个服务器解决问题，登录服务器之前需要知道它的主机名、IP 地址、MySQL 版本，以及主操作系

467

统信息，管理员需要记下所有重要的信息。图 13-8 是 Enterprise Dashboard 上的服务器详情的样例。

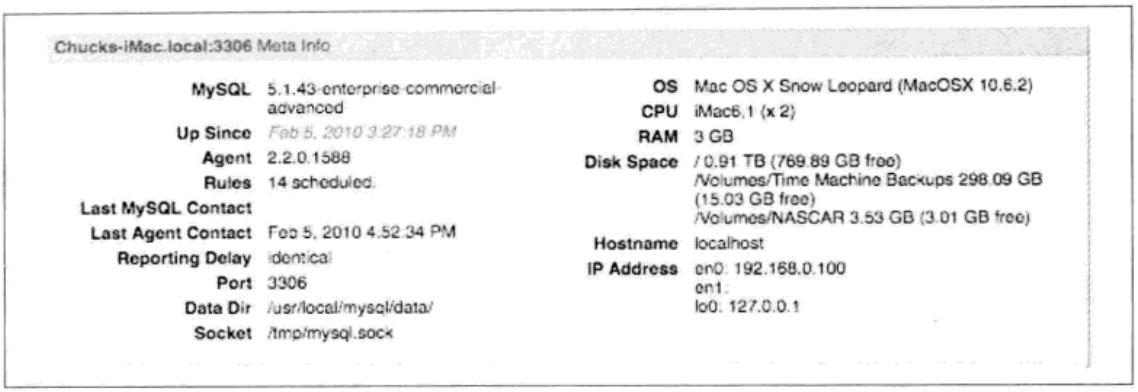


图13-8：服务器详情

468

复制的详情

Enterprise Dashboard 的复制选项卡包括参与复制的所有服务器的列表。这些信息以列表的形式呈现，与 MEM 中的所有列表一样，你可以单击每个项以获取更多的信息。图 13-9 显示了复制的详情报告的一个示例。

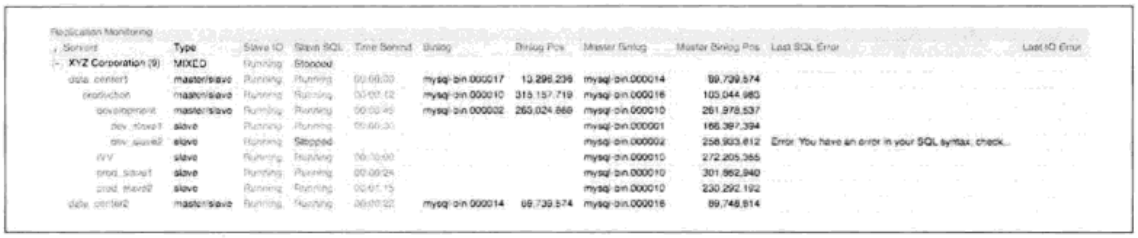


图13-9：复制的详情

请注意，列表中的项目是按照拓扑结构分组的（如“XYZ Corporation”，你可以重命名），这些项目包括拓扑类型、服务器执行的角色，以及复制相关的重要统计信息（包括线程状态、比 Master 滞后的时间、当前二进制日志、日志位置、Master 日志信息和最近发生的错误）。

本例我们看到 *dev_slave2* 服务器出了问题，执行查询时出错。这是一个如何快速了解复制拓扑的好例子。这个列表显示了 Master 和 Slave 的分组层次，即在一个组下面显示 Master，隶属于 Master 的 Slave 显示在 Master 下面。从图 13-9 很容易发现开发服务器有两个 Slave，即 *dev_slave1* 和 *dev_slave2*，而开发服务器是产品服务器的 Slave。将复制的所有信息保存在一个地方，就不需要在每个服务器上单独执行烦琐的监控复制任务。

Advisors

advisor 中实现的最佳实践使得所有的警告和图更加有益。从 Enterprise Dashboard 上的 Advisors 选项卡可以查看所有活动的 advisor（并可以创建你自己的 active advisor）。图 13-10 列出了白金版中的默认 advisor 列表。

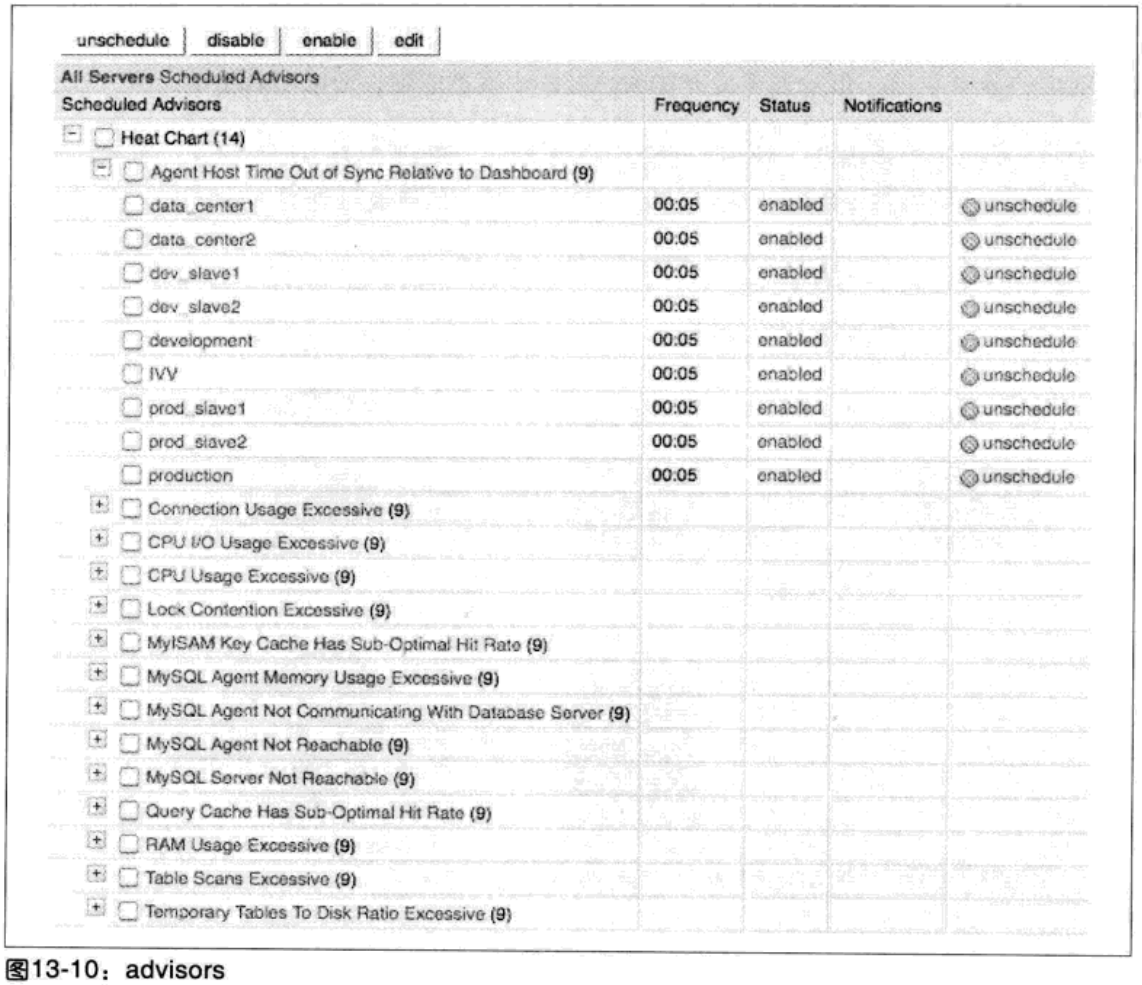


图13-10：advisors

图 13-10 显示了网络中某个服务器的活动 advisor，你可以启动、禁用或取消任何 advisor（取消是指删除收集到的数据）。

也许该页中最有用的功能是添加自己的 advisor。这个功能允许你根据特定的需求扩展 MEM。另外，从手动监控方案转到监控工具上时，它还可以为你提供一个最有用的功能，即保留你的辛勤劳动成果。

例如，如果你创建了一个用于监控自定义应用的报告机制，你可以为它创建一个 469

advisor，并将警告添加到 Enterprise Dashboard。具体如何添加新的 advisor 和警告，在 MySQL 企业网站中的 MySQL 企业监控手册中有详细描述。这个自定义功能是 MySQL 企业版的最强大但却未被充分利用的功能之一。

查询分析器

MySQL 企业版的最新功能是查询分析器。专业的数据库管理员早就知道单独使用 MySQL 查询分析器，但是直到最近查询分析器才被集成到企业版工具集中。

查询分析器的工作原理是：使用 MySQL 代理拦截并处理 SQL 命令，然后将这些 SQL 命令传递给本地服务器执行。它可以记录统计数据，以便随时查看这些信息。查询分析器还支持 advisor，这个 advisor 对慢查询发出警告。图 13-11 显示了 MySQL 代理拦截查询并向 MEM 报告统计信息的概念图。

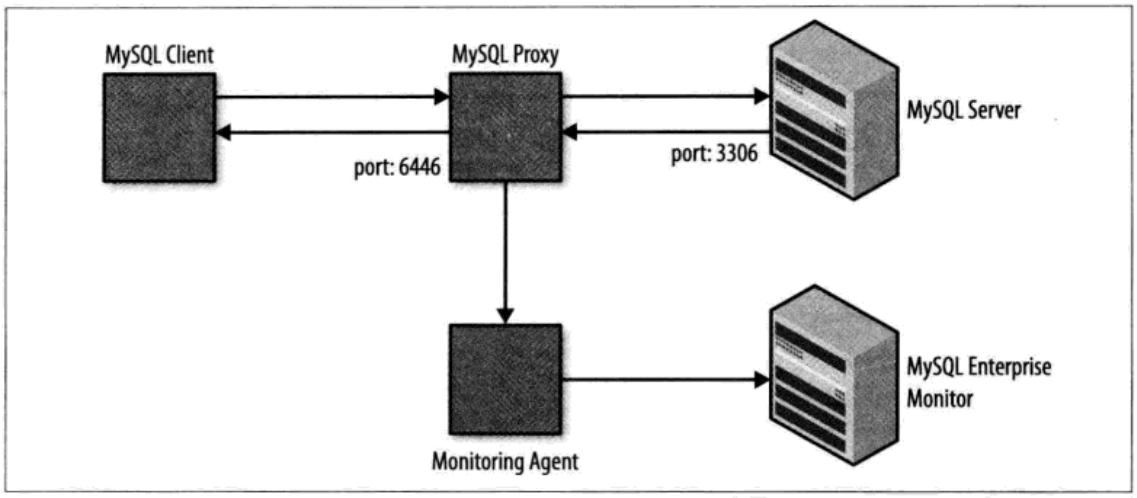


图13-11 使用MySQL代理为查询分析器收集数据



查询分析器在自定义的 6446 端口（默认）运行，而且可能会带来性能延迟。因此，你应该仅在查找问题时启用它。

470

为了给查询分析器收集数据，必须将客户端连接到 MySQL 代理的端口上，并将代理的端口配置为 6446。如果在 Enterprise Dashboard 上没有看到任何查询的报告信息，首先确定是否能连接到该端口。

虽然 MySQL 代理可能会导致查询执行吞吐量上轻微的延迟，但是分析不良的查询所带来的好处超过了这个延迟代价。你可能只想在某些开发或试验机器上使用查询分析

器，而不在产品服务器上使用。查询分析器和 Enterprise Dashboard 的一个好处在于工具之间的紧密集成。如果你有一个慢查询的警告，或者需要深入研究 CPU 利用率或其他 MySQL 统计信息的报告，将看到查询分析器页面（单击查询分析器选项卡可以直接进入这个页面）。图 13-12 是 Enterprise Dashboard 的查询分析器页面的例子。

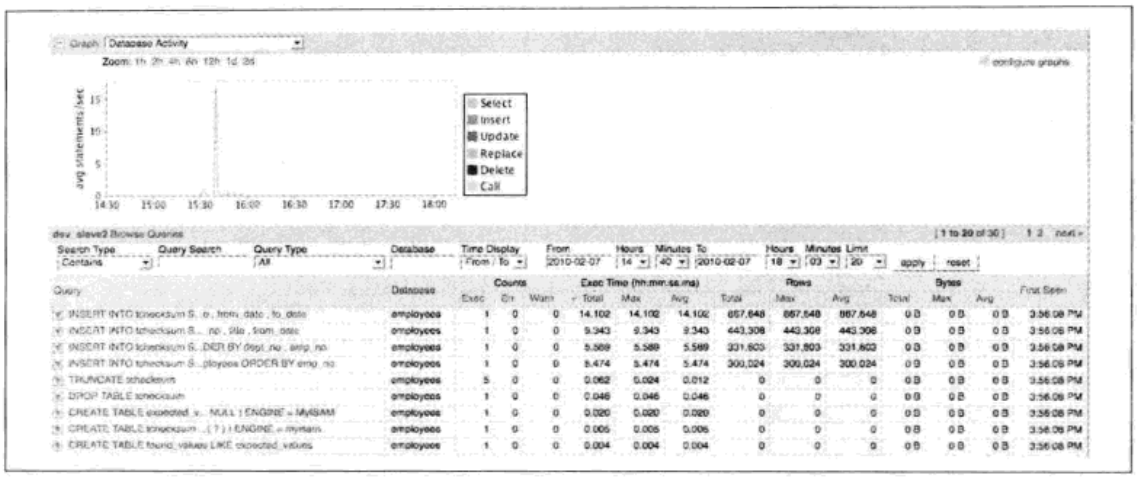


图13-12：查询分析器展示

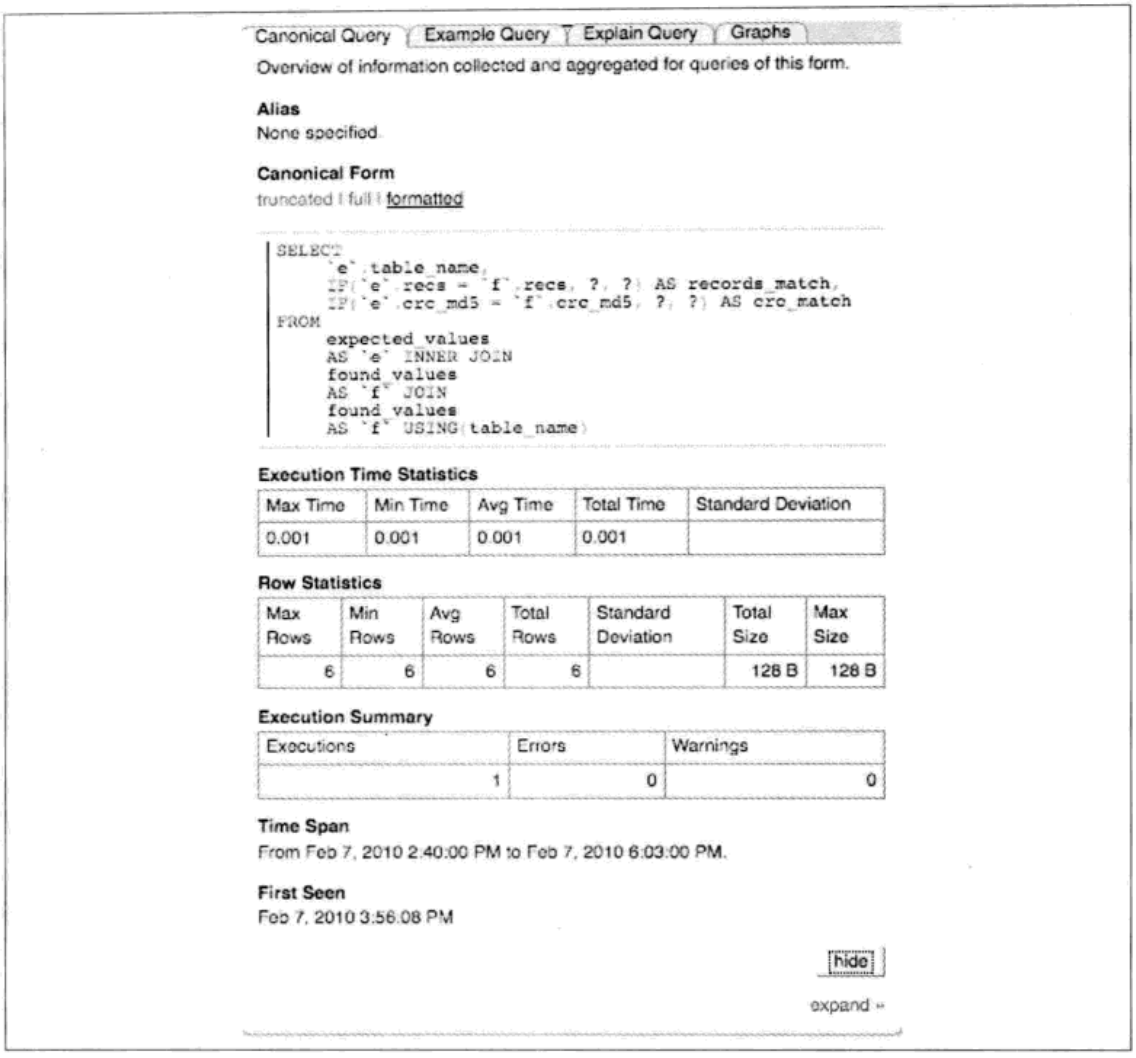
查询分析器页面的左边显示了服务器列表。单击某个服务器，将看到该服务器上执行的查询的列表，按查询运行的时间降序排列。还可以使用顶部的图缩小时间范围，以查看特定时间段内的查询。 471

单击列标题可以排序，便于发现重复的查询。你可以按查询运行的时间、查询语句和操作的数据量等对行进行排序。

单击任何一行可以获取查询的细节信息。图 13-13 显示了一个查询报告的典型例子。与其他报告一样，这里有更多的详细信息，包括 Example Query 选项卡中的实际查询，Explain Query 选项卡中 EXPLAIN 命令的输出结果（如果启用了这个选项），以及 Graphs 选项卡中关于执行时间、执行次数和返回行数的图等。

这个报告显示了捕获到的查询详情，包括查询的规范形式（即书面查询的图形表示）及其执行详情，例如执行时间和返回或受影响的行。 472

我们再一次看到了这样的监控工具：它可以为 MySQL 问题诊断节省大量的时间。MySQL 企业版的查询分析器组件是个重要的监控工具，有助于维持服务器的正常运行和查询的高效执行。



Canonical Query | Example Query | Explain Query | Graphs

Overview of information collected and aggregated for queries of this form.

Alias
None specified.

Canonical Form
truncated | full | formatted

```
SELECT
     `e`.table_name,
     IF(`e`.recs = `f`.recs, ?, ?) AS records_match,
     IF(`e`.crc_md5 = `f`.crc_md5, ?, ?) AS crc_match
FROM
     expected_values
     AS `e` INNER JOIN
     found_values
     AS `f` JOIN
     found_values
     AS `f` USING(table_name)
```

Execution Time Statistics

Max Time	Min Time	Avg Time	Total Time	Standard Deviation
0.001	0.001	0.001	0.001	

Row Statistics

Max Rows	Min Rows	Avg Rows	Total Rows	Standard Deviation	Total Size	Max Size
6	6	6	6		128 B	128 B

Execution Summary

Executions	Errors	Warnings
1	0	0

Time Span
From Feb 7, 2010 2:40:00 PM to Feb 7, 2010 6:03:00 PM.

First Seen
Feb 7, 2010 3:56:08 PM

hide

expand »

图13-13：典型的查询报告

MySQL 企业版和云计算

MySQL 企业工具在云计算环境中也很有用。从数据和服务器实例的持久性来说规则相同，只是要为 MEM 服务器设置一个固定的 IP 地址。重复启动和停止 MySQL 实例将不会造成不良影响，但是更改主机名及某些重新配置可能导致监控代理停止报告任何信息。一般截断 *mysql.inventory* 表可以解决这个问题，但是最好为所有的服务器使用相同的主机名和 IP 地址。

在商业提供者的云上运行 MySQL 企业版有一个明显的好处：只需要为计算和存储付费。云内的数据传输通常是免费的，或者其费用比将数据从云中导入和导出低得多。

更多信息

本书不对 MEM 做全面而详细的描述，这些信息可以从网上获取。要想了解更多关于自定义 advisor 的信息，请参看企业订阅门户网站上的 MySQL 企业监控手册。此外，常见问题的解决方案参看门户网站上的知识库。

小结

MySQL 服务器已经成为最流行的开源数据库系统。它的用途相当广泛，任何组织机构都可以使用它，从一个为用户提供信息的独立的 Web 服务器，到在线联机事务处理系统（OLTPS），再到高可用的不断扩展的大规模数据中心。MySQL 可以处理所有这一切。而 MEM 将 MySQL 扩展到大型应用上，并保存了 MySQL 的最佳性能和可靠性。

如果你需要高可用性并希望建立最好的和最可靠的基于 MySQL 的数据中心，那么就应该考虑购买 MySQL 企业白金版。或许你还有其他的选择，但是没有哪个比 MySQL 企业版更专业，也没有哪个能够以便宜的价格为你提供成熟的 advisor 和查询分析器工具。

Joel 准备好了。他已经把提案发出去了。他建议购买 MySQL 企业版来管理公司所有的服务器。Summerson 先生以削减开支闻名，所以 Joel 已经准备好捍卫自己的建议，决定不放弃，直到 Summerson 先生听进他的建议为止。 473

每当听到脚步声临近时，Joel 都会看看门口。他知道老板可能随时过来。一阵脚步声响起，Joel 再一次紧张起来。但 Summerson 先生连看都没看一眼就快速走开了。

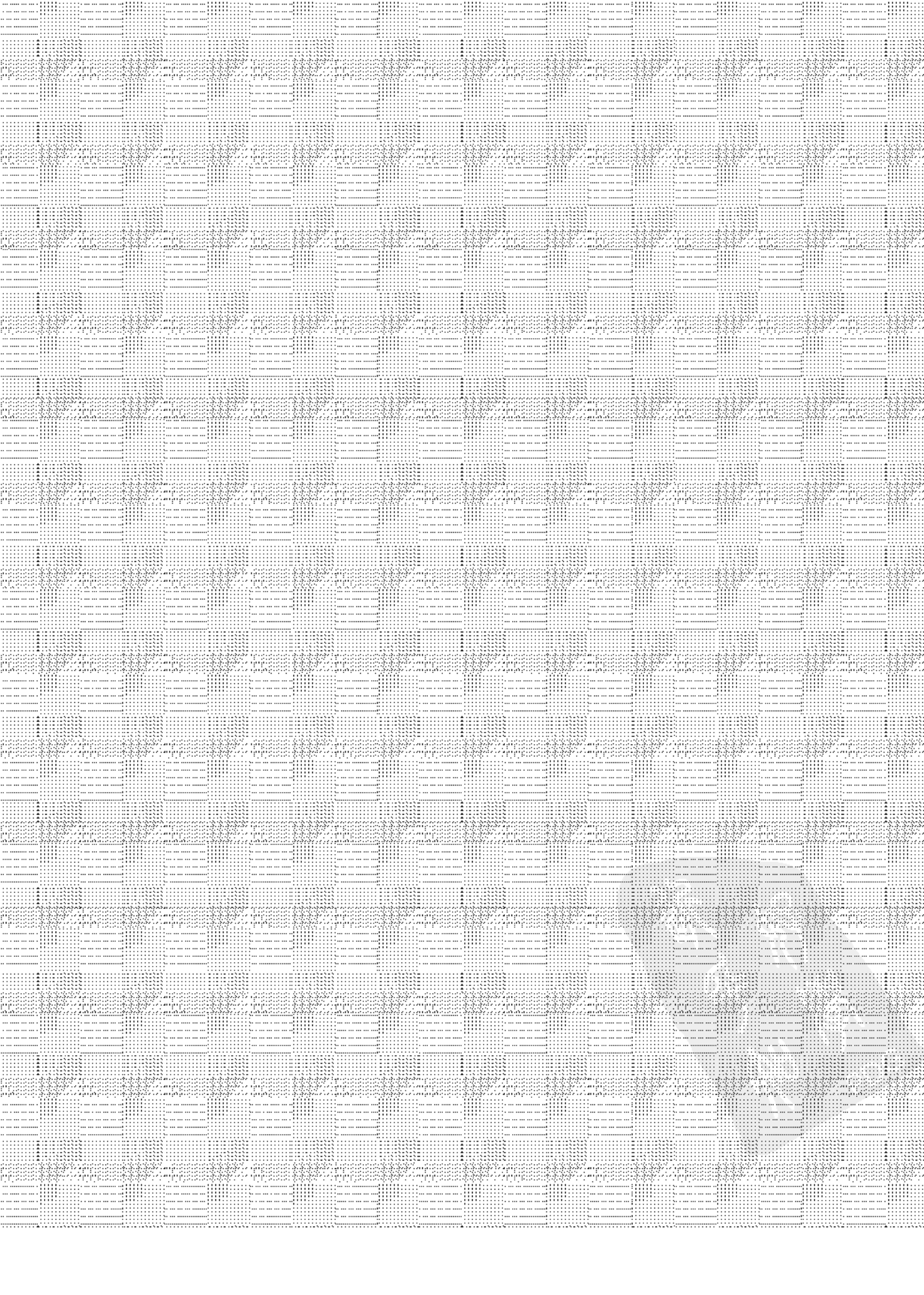
“请等一下。”Joel 低声说。

“Joel，我喜欢 MEM 这玩意。我希望你评估一下这个整体成本和……”

Joel 打断老板的话：“我已经给你发了一份报告，Bob。”

Summerson 先生扬起眉毛说：“把它发给我，我看看是否可以把它列入明年的预算中。”然后 Summerson 先生就离开了，走向下一个受害者。

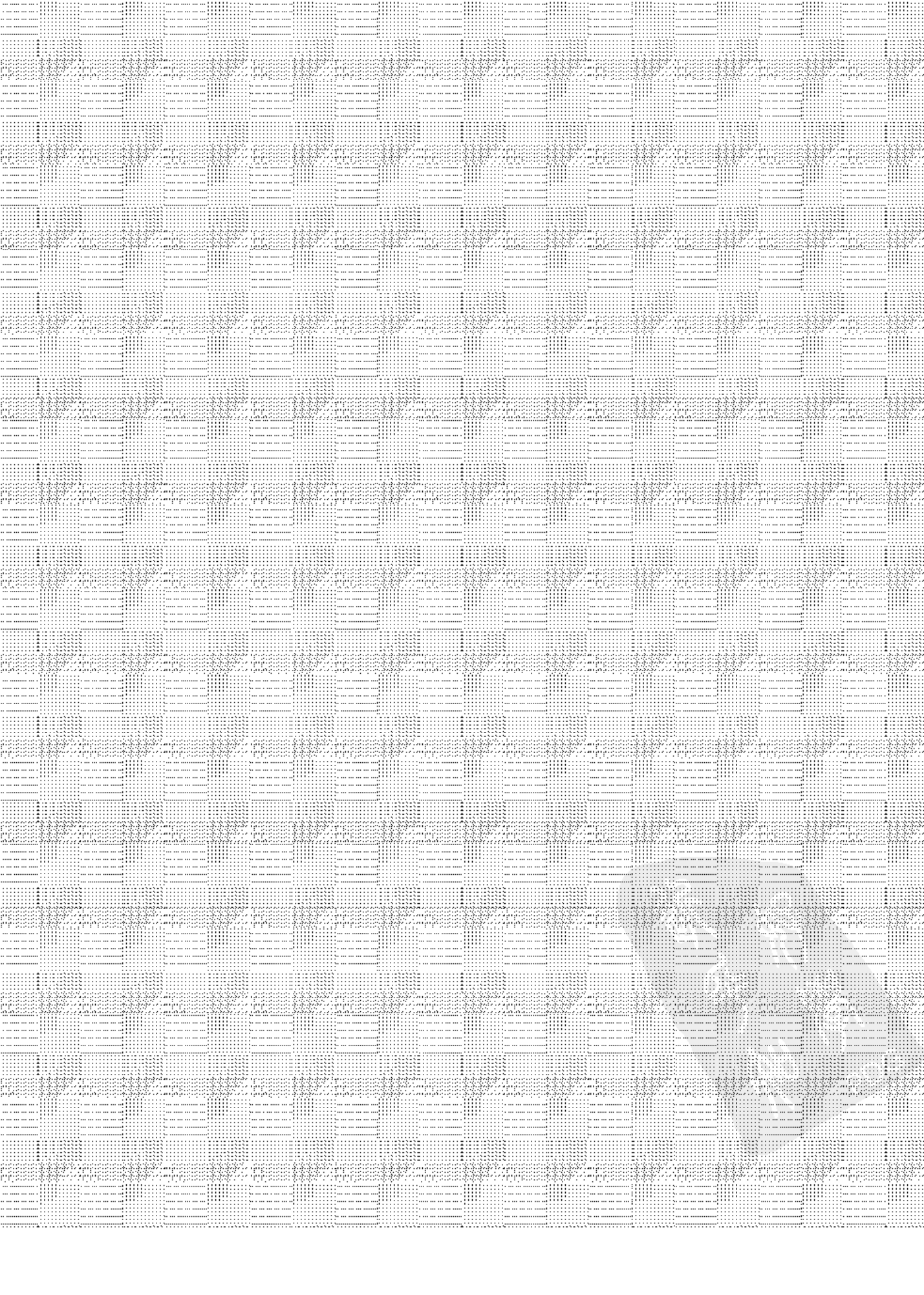
Joel 回到座位上，得意地把双臂叉在胸前。“我看那是赞许吧。”他说。



第三部分

高可用性环境

云计算和 MySQL 集群提供了满足高可用性需求的新机遇。本书的这一部分将介绍这些主题。



第14章 云计算解决方案

Joel 稍稍关上办公室的门，把夹克衫挂在门后的钩子上。这时听到一阵连续敲门声，他急忙一边说“请进”，一边把门拉开，然后走回桌边。他转过身面对着来访者，他知道这家伙是谁。

“Summerson 先生，早啊。”

“早，Joel。那个关于高可用性和横向扩展的报告做得不错，我特别希望你能给如何提高部分产品的生产量提一些意见。”

“谢谢。”Joel 屏住了呼吸，等着他早知道一定会来的任务。

“昨晚董事会和我签了一个合同，为一个新客户定制我们的一个产品。合同上的字迹还没干，所以我不打算深入细节，但可以说我们需要很多高可用配置的新服务器。当然他们会使用 MySQL 作为数据库组件。”

Joel 试图记起他曾经读过的关于 MySQL 高可用性的所有细节，想知道搭建一群服务器需要多少钱。老板继续说着，打断了他的思路，“然后还有一些关于负载均衡的事情。”

一段不安的停顿之后，Joel 说道：“好的，先生。”

“问题是我们没有足够的资源来购买一群服务器，合同上是 6 个月的服务期。董事会不会同意在这些合同过期后可能就不再需要的新硬件上花大量的钱，更别提损害我们的利润空间了。”

Joel 不知道该说什么，所以他只是等着。

“所以，我们想要你组建一个基于云的解决方案。”说着，Summerson 先生拍了拍 Joel 的肩便离开了。Joel 站了一会儿，然后走到桌子后面坐下来。

Joel 认为自己了解新技术，但他想云计算这东西就像它的名字一样空洞。他捡起卷角的 MySQL 书，翻到下一章。“好吧，看看这个。”他读了起来。

478 今天的经济需求带来了新的挑战，也为信息基础设施规划带来了潜在的新解决方案。每次需要添加计算能力的时候，组织不再简单地购买更多的硬件。过去的十年里用于计算的硬件成本下降，组织的利润空间也下降了，特别是最近。

因此，组织必须做出一个财政负担更强大的决策，寻找最实惠的服务和工具扩大生产线，增加客户基础，同时减少成本、提高利润。所有这些终究都需要钱。

实惠的计算解决方案的需要，引导技术供应商创建一种“pay-as-you-go”的理念来使用计算机，这使得组织在临时安排的基础上可以购买计算和数据的使用权。这就是计算机科学家们所描述的“云计算”的本质。

什么是云计算

云计算是经常会被误解的短语之一，更不幸的是，它有几个（常常是冲突的）定义。有人马上说它是一个描述已有技术的漂亮口号，有人则认为它精炼了学术和科学（有时候也有社会）的方方面面，也有人坚称云计算是未来的信息技术。

因此，有人断然说云计算只不过是网格计算，也有人说云计算可以代表整个 Internet。这两种观点都有缺点。还有人详细解释了“as a service”概念的各个方面，并使用它们来定义云计算。幸运的是，这些更细致入微的看法更接近真理。

云计算本质上是一组技术的再度融合，包括网格计算、虚拟化结合应用程序接口（API）和工具，并提供了虚拟化环境的访问接口和工具。在“*Cloud Computing Architectures*”（O’Reilly, *http://oreilly.com/catalog/9780596156374*）这本书中，George Reese 说“根本没有任何新的技术构成云计算。”这是一个发人深省的观察，一些专家和营销人员不愿面对它。然而，通过这种包装形式，像亚马逊这样的巨头已经以全新的方式使用已有的技术，而且取得了显著的效果。

James Governor 在论文“15 Ways to Tell It’s Not Cloud Computing”中强烈声明了什么是云计算、什么不是云计算。套用 Governor 的观点，他明确了云计算不需要很长的时间

来解释，没有复杂和陡峭的学习曲线，不是孤立的，不需要使用专用连接，也不要求你购买硬件。不管你是否赞同他的观点，他迫使公司的赞助者需要考虑产品的购买量，从而小心谨慎地加入“云”的概念以重新投放市场，因为云不是大多数人所想象的那样。 479

术语“云计算”来源于描绘大型网络（云）中托管的资源的概念图。我们使用云的符号是因为资源的实现（如硬件、操作系统等）都是隐藏的，与什么服务或者供应商几乎不相关，它只是一个简单的可供使用的服务。因此，与网关、路由器和服务器不同，你会看到资源作为服务被提出。资源的消费者并不关心服务是如何实现的，最重要的是服务能够解决需求，并在需要的时候可用。

并不是说“云计算对很多人来说意味着很多事”，美国国家标准技术研究院（NIST）这样定义云计算：

> 云计算是一种方便的模型，提供对可配置计算资源（如网络、服务器、存储、应用和服务）共享池的按需访问，可快速配置，需要最少的管理和服务提供者交互。这个云模型提升了可用性，由五个基本特征、三个服务模式和四个部署模型组成。

大多数云研究者定义了云计算必须包含的几个特点：

- 按需自助服务
 客户可以在点对点需求的基础上选择他们所需要的没有第三方供应商或卖家介入的服务。
- 广泛的网络接入
 资源通过现有的网络功能可获取。
- 资源池
 多用户共享供应商的硬件（如多租户共享模型）。
- 快速弹性
 客户能够手动或自动地快速伸缩资源。
- 测量过的服务
 提供主动或被动的资源监控和管理服务（参见第 10 章）。

三种服务模式如下。

- 基础设施即服务（*IaaS*）
 资源通过完整硬件或者操作系统平台的虚拟实例提供。客户端可以按需添加虚拟的计算资源（如服务器、均衡负载器）。因此，信息基础设施的组件以组件或中间件的形式提供。客户可以访问和控制这些资源（如客户可以管理一个分配服务器）。
- 平台即服务（*PaaS*）
 一个 API 允许客户端创建特定设计的应用程序以运行在供应商的硬件（平台）上。 480

供应商提供托管环境和编程工具，允许客户构建特定环境下的解决方案。

- 软件即服务（*SaaS*）
 运行在提供商的硬件上的软件，以应用程序的形式作为资源被提供。客户只能看见软件的接口，就像一个桌面应用一样。硬件、操作系统等都被供应商完全隐藏和控制。这是包含在当前云定义中的最古老的模式，多年来称之为应用服务提供商（*Application Service Provider*，*ASP*）。

部署模型是指最终解决方案的可用性或可达性，包括：

- 私有云
 资源的访问仅限于客户。
- 社区云
 资源的访问在一个或多个客户之间共享。
- 公有云
 资源的访问对大众可用。
- 混合云
 这是一种由两种或更多其他模型组合的基础设施。典型的情况是，私有和公共资源之间可以互相沟通。

关于 NIST 的云计算观点的完整讨论见 *http://csrc.nist.gov/groups/SNS/cloud-computing/*。

云架构

在这部分我们将简单介绍一下云计算中常用的底层技术。你会发现大多数云计算解决方案部署采用的都是这些技术（可能不全包括）：

- 虚拟化
- 网格计算
- 事务计算
- 弹性
- 软件库

481

虚拟化

虚拟化有多种形式。如果你曾经用过 Sun 的 VirtualBox 或者微软的 VirtualPC，你就已经用过一种形式的虚拟化了。本质上，这种技术在概念化计算硬件模型的基础上创建了一个伪平台。例如，可以在 Linux 机器的 VirtualBox 内运行微软的 Windows 操作系统。VirtualBox 在 PC 上创建了每个组件的软件模型。这些构成了 Windows 启动和运行的基础，就像真实的硬件一样。

这仅仅是虚拟化的一种形式。除可以优化启动、执行和管理实例外，大多数 IaaS 解决方案中使用的虚拟化需要你使用预打包的机器（称为映像），每台虚拟机器称为映像的实例。例如，Amazon 的云使用开源的 Xen 虚拟化技术，允许虚拟硬件（如 CPU 数量）的扩展、容错等其他好处。

而且，有些供应商让你修改现有的映像以便定制满足你的需要的机器，可以使用供应商自己提供的工具或者使用特定格式的机器描述来修改映像。这样，如果你想从一个供应商迁移到另一个供应商就会出现问题：你的映像是不可移植的。在这种解决方案中，你需要在投入大量时间和精力之前检查供应商关于映像和定制映像的文档。

网格计算

在计算能力有限的日子，解决复杂分析问题或科学问题的需求很大，一种允许程序在相互连接的机器社区内共享额外的计算能力的技术就诞生了，称为网格计算。它将问题分解为较小的可由其他机器处理的计算单元，然后检索结果并将其关联到一台机器上。

允许机器之间通信的关键技术是复杂的排队机制。这种排队机制就像工作流一样，可以这样来描述：数据管理器将 job 和数据委托给 Slave 机器，然后从那里读取结果。这需要设置一个或更多的排队机器，它们将处理中的 job 发送到任何相连的计算机上并将结果返回给数据管理器。当用户想要参与到网格计算程序中的时候，他首先要将自己的电脑连接到一个排队机器上，并发出处理包（job 和数据）的 pull 消息。他的机器执行这个 job，然后把结果发给排队机器。图 14-1 给出了这一过程的简单示例。

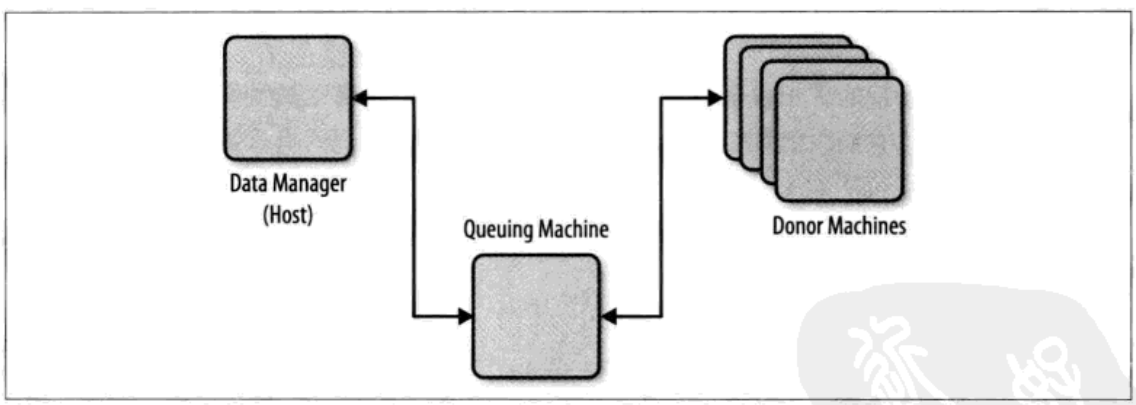


图14-1：网格计算工作流

排队系统也存在于云计算中。因此，将现有的网格计算解决方案迁移到云计算或者在云计算中构建一个新的网格计算解决方案是可能的。这就是为什么有些人坚持说云计算只是简单的网格计算加上虚拟化的原因。但是你会看到，除了这两种技术以外，云计算还

482

包含更多东西。

事务计算

数据库用户都比较熟悉事务计算：多个数据段可以在单个事务中被同时处理，并和其他数据相互关联。这种想法就是定义了一个job，它包含特定数据并在这些数据上单步地(即事务)执行一些操作。最好的网格计算解决方案使用这种概念来保证结果的正确交付。然而，云计算稍微有点复杂。具体来说，大型事务应用要运行很长一段时间，而网格解决方案只需很少的执行时间。

幸好在云计算环境中构建一个事务计算系统是有可能的。为此，我们必须要保证计算资源长寿，并提供允许数据分段和并行处理的机制。如果你在想“嘿，这听起来似乎像服务器农场”，那么你是对的。大多数云计算供应商提供虚拟资源来支持事务计算解决方案，包括负载均衡器、持久化实例和网络资源的永久分配。

弹性

我们用术语“弹性”来描述已经成为商品的抽象网络或系统资源。例如，Amazon允许你将给定的IP地址应用到它的云环境中的任何服务器实例上。这在事务系统中是很重要的，你需要一个回应特定地址的服务器池。这是很好的服务器虚拟化，所以它们可以在云中的任何地方运行，而你必须有一个方法来保证IP地址保持不变。

483

在这种情况下，IP地址就成为一个弹性资源，你可以将它分配到任何实例，而不是捆绑在一个特定的机器上。同样，磁盘资源也是弹性的，你可以在一个磁盘资源上存储数据，云中的任何正在运行的实例都可以访问它。

弹性解决了正在运行的虚拟机器的硬件配置问题。机器可以真正做到即插即用，你可以轻易地创建和毁掉它们。例如，可以从一个运行某种操作系统的机器切换到另一个开发过程中的机器上（可能有一些细微的改变），并仍然可以访问同样的数据而不需要重新创建一个新的数据库。

软件库

你可能想知道所有这些技术是怎样联系在一起的，在动态环境中怎样与各种资源协同工作。答案是大多数云供应商都有一套特定的工具来创建和管理云中的资源。

例如，Amazon有自己的API工具来管理你的资源、创建实例、创建卷（即磁盘对象），以及更多的事情。这些工具包括用于管理云资源的Amazon EC2 API工具，以及用于创建和修改机器映像的Amazon EC2 AMI（Amazon机器映像）。

同样，微软 Azure 的 .NET 开发扩展插件可以让你构建自己的云应用，并在 Azure 云中运行。然而，和 Amazon 的相似之处只有这些，因为微软 Azure 环境还需要你利用这些库创建自己的应用，而 Amazon 则不需要。

因此，软件库成了让所有这些已有技术联合在一起构成强大的云计算环境的黏合剂。

云计算是一种经济的选择吗

分析家和专家们对这个问题有各自不同的看法。最起码这要视情况而定，取决于你使用的是哪个云提供者，需要多少服务器（以计算时间来衡量），需要多大存储空间，以及要使用多长时间。

有一个比较研究给出了在典型的电子商务场景下云解决方案和传统解决方案（即自己购买硬件）五年内的成本对比，结果显示云计算只稍微占优势。

你可能觉得这说明了云计算并不能节约很多成本，从表面上看你是对的。但是，这项研究的细节表明了传统方案的初始投资非常高。五年后这个组织的确会拥有自己的硬件（或者进行折旧摊销），而云中的硬件不是这样，因为它不是成本的因素之一。

也就是说，当使用基于云的解决方案时，不需要设备升级所带来的经常性费用。这项研究并没有在比较中包括这个费用，如果包括，可能这五年来的费用就会远远低于传统方案。 484

为了回答云计算是否经济这一问题，你必须做点功课。每个组织每个项目的成本因素都各不相同。分析这种成本的最好办法是确定（或估计）服务器的数量、需要存储多少数据、多少数据可能在云中移动，以及你需要的传统特性（VPN，负载均衡等）。尤其是需要检查云计算供应商的计费组件，然后在这些参数的基础上估计成本。一旦做了这些，就可以确定传统方案（包括维护和升级）的成本，然后做出一个公平的比较。

很多客户端采用云计算并不是为了省钱，而是看中了它的灵活性。我们将在以后的章节中讨论为什么这会是一个有价值的解决方案。

但是，有些组织认为云计算的应用要么因为政策原因要么由于担心害怕而无法得以发展。例如，有些组织不允许私有数据存储在非他们自己拥有的系统上，而且（令人信服地）一些组织外的管理员（如云供应商的雇员之一）可能会获取对这些数据的访问。如果你发现自己面临这样的问题，应该联系云供应商，跟他讨论你的顾虑并权衡相应的风险。克服这一缺陷的一个方法是将你的数据和云中使用的公共数据分离开来。而且，拿 Amazon Web Services（AWS）来说，云供应商允许你的云实例分离，或者通过 VPN 将它们连接到自己的 IT 基础设施上（详见 *http://aws.amazon.com/vpc/*）。

云计算实例

既然你已经对什么是云计算有了很好的了解，现在让我们来看看云计算可以做些什么。所有组织方式都会发现云计算令人兴奋的新用法，其中包括刚成立的公司为进入市场寻找廉价的门槛，研究者需要有限时间内的计算能力，信息技术管理者感觉预算紧缩但必须要满足用户的需要。本节就讨论使用云计算的一些常见实例：

- 传统 web 服务
 云资源向互联网用户提供内容和应用。

485

- 共享服务
 不同用户可以共享在云中运行的一个或多个应用。例如某个应用允许合作伙伴之间协作和共享数据（如供应链）。
- 企业横向扩展
 这使得云解决方案中的应用可以快速扩展，并连接到企业中去。
- 爆发云
 这使得用户使用临时资源快速解决突发的短期计算任务。
- 研发
 这使得开发人员可以开发多个系统和应用配置，而不需要专用硬件。

可见，使用云计算的方法很多，每天也在不断地发现更多的用途。到目前为止我们在这一领域所看到的仅仅是一个开始。

云计算的好处

云计算的潜在好处包括：

- 减少了运行时间和响应时间
 采用网格或者横向扩展技术，就有可能极大减少任务完成时间，甚至大幅减少数据访问时间。用基于硬件的方法也可以达到类似的效果，但那样会产生过高的投资成本。云方案允许产生需要的足够多的机器实例，且只要为实际使用的那部分付钱。
- 最小化基础设施风险和维护
 硬件故障不再是你的责任了。供应商拥有和维护云系统中运行的机器，所以在服务提供方面，不需要大量的人力投入或者投资。
- 降低了初始投入成本
 你只需为实际使用的部分付钱，不再需要为一个可能用不上的庞大的基础设施做预算。最棒的是，可以动态扩大基础设施，更不错的是，还可以缩小规模而不需要摊销硬件或声明其过剩。

- 加快了发展的步伐
 初始投入成本降低了，只需为你需要的部分付钱，这意味着你可以用比以前（在自己或者服务提供商的硬件上）更少的投资开始一个新的应用。这对于新成立的公司有副作用，使他们在发展过程中更早地参与竞争。

当然，每个好处都对应着一个缺陷。云计算的潜在风险包括：

486

- 服务失效
 服务等级协议（SLA）在云计算领域的定义不明确（或者不存在），如果你的底层服务不可靠，你几乎没有追索权。
- 潜在的成本失控
 如果系统处于异常的重负载下，可以通过纵向扩展来满足它，可能可以满足需求，但这样会带来很高的使用成本。
- 缺乏特征
 有时候你可能需要在自己的架构或应用中实现一个供应商不支持的特征。
- 安全风险
 你正在跟其他用户一起共享机器，软件问题可能使数据泄露或被盗。

云计算供应商

只要有新技术出现，就不可避免地涌现出一批供应商、产品和服务，都以这样或那样的方式宣称提供了这项新技术。云计算也不例外。成百上千的供应商提供了一系列东西，从专用硬件、软件服务、平台到可立即投入使用的投资组合。如果你使用前面的“什么是云计算”一节中给出的 NIST 定义，你会很快发现很多供应商并没有满足定义中的所有原则。

然而，仍然有很多供应商努力满足云计算的完整定义。下面列出了前十名供应商，以及每个供应商提供的解决方案的简略描述：

- *3Tera*（*http://www.3tera.com/*）
 专注于快速横向扩展能力的 IaaS 提供商。
- *Akamai*（*http://www.akamai.com/*）
 专注于 Web 数据管理的 IaaS 提供商。
- *Amazon*（*http://aws.amazon.com/*）
 提供虚拟化 SaaS、PaaS 和 IaaS 的存储解决方案的云计算供应商。
- *Enki Consulting*（*http://www.enkiconsulting.net/*）
 专注于虚拟私有数据中心解决方案的 IaaS 提供商。

- *IBM Blue Cloud*（*http://www.ibm.com/ibm/cloud/*）
 提供虚拟化 SaaS、PaaS 和 IaaS 解决方案的云计算供应商。
- *Joyent*（*http://www.joyent.com/*）
 专注于大型企业需求的 IaaS 提供商。
- *Layered Technologies*（*http://www.layeredtech.com/*）
 PaaS 和 IaaS 提供商。
- *Rackspace*（*http://www.rackspace.com/*）
 专注于为 web 应用提供主机服务的 PaaS 厂商。
- *Salesforce.com*（*http://www.salesforce.com/*）
 专注于共享客户关系管理（CRM）解决方案的 SaaS 供应商。
- *Terremark*（*http://www.terremark.com/*）
 IaaS 提供商。

487

甚至 Apple 也宣布将其服务和产品扩展到 Internet 可访问的 pay-as-you-go 解决方案中的计划。尽管没有明确定义，也没有作为一个云计算方案被提出来，但它也可能成为 Google Docs 和微软在线 Office 套件计划的竞争者。显然云计算背后的概念在信息技术产业的很多大型企业中有着深远的影响。

考虑到 AWS 的流行程度、种类繁多的服务、成熟度和复杂性，本章我们选择重点讨论一下 Amazon 的云计算产品。当你选择云计算供应商的时候，鼓励你们考虑和权衡你的特定需求。但是，你会发现 Amazon 的解决方案迅速变得十分常见。

Amazon Cloud：另一个 Xerox？

正如我们很多人提到任何办公复印机都是“Xerox 机器”，并使用相关动词“xeroxing”一样，很多技术权威人士都以 AWS 提供的服务来描述云计算（或简称“云”）。只有时间和产业应用模式会告诉我们 Amazon 解决方案是否会成为所有解决方案的判定标准。然而，我们常常听到云计算被定义为具有弹性——因为 Amazon 选择将它的 IaaS 产品命名为“弹性计算云”（EC2）而流行起来的名词术语。

AWS

Amazon 提供了一组开发工具和解决方案，称为 AWS。它们的全称为“Amazon Cloud”，这个云提供的产品和服务包括计算服务（云）、内容交付、数据库系统支持、电子商务解决方案、消息、监控、网络、供应商的支付和计费解决方案、云存储解决方案、AWS 产品的支持服务、Web 流量管理和劳动力的软件解决方案等。Amazon 定期添加新产品，

所以你阅读这本书的时候，上面所列出的 Amazon 的产品和服务的列表将会大大增加。

Amazon 的产品基本上是收费的，但用户在需要消费的时候才需要为服务付费。 488

下一节将讨论 AWS 中的技术，并简单介绍 Amazon 云计算的使用方式。

技术简要概述

由于有关 AWS 的技术繁多，我们仅重点介绍关于构建可靠数据中心的领域。当然，想要探索所有 AWS 产品的具体细节，到 *http://aws.amazon.com* 主页的顶部单击 Products 标签页即可。

下面的所有技术都是建立在简单 Web 服务之上的，这样可以简化应用程序的构建，通过 REST Web 接口使任何工具应用之间都可以相互通信。

- *Amazon* 弹性计算云（*EC2*）
 它和 Amazon 简单存储服务（S3）一起构成了 Amazon Cloud 的核心。这是云方案的核心技术，管理虚拟计算资源。
- *Amazon* 弹性 *MapReduce*
 使用 Hadoop 框架为数据密集型任务提供环境。
- 自动伸缩
 提供基于自定义参数的解决方案的自动伸缩能力，这是建立高可用云方案的关键特性。
- *Amazon CloudFront*
 内容管理服务，在各种不同的地理位置为用户提供静态内容和流内容（有时候称为主动内容）。
- *Amazon SimpleDB*
 提供基本的非关系型数据库存储和检索。
- *Amazon* 关系数据库服务（*RDB*）
 Amazon 对 MySQL 数据库系统提供的东西。可以使用这项服务来代替创建和管理自己的 MySQL 安装。
- *Amazon* 实现 *Web* 服务（*FWS*）
 提供一组绑定的电子商务工具——和现在著名的 Amazon 商业网站使用的工具相同。
- *Amazon* 简单队列服务（*SQS*）
 网格计算解决方案中使用的队列消息服务。
- *Amazon CloudWatch* 489
 提供监控所有 Amazon 云资源的能力。

- *Amazon* 虚拟私有云（*VPC*）
 它是最近一个令人兴奋的功能，允许企业在 Amazon 云中使用 VPN 来扩展其基础设施。VPN 中分配的资源与内部基础设施通信，就像它们在相同的内部网络中一样。这对必须要快速提高其计算资源的组织来说是必需品。
- 弹性负载均衡
 云服务的又一个关键组件，在解决方案中提供网络流量的负载均衡能力。
- *Amazon* 灵活支付服务（*FPS*）
 提供建设慈善基金会和电子商务网站所需的支付处理工具的开发库。
- *Amazon DevPay*
 为网上零售商提供在线计费和会计服务。
- *Amazon S3*
 Amazon 云的另一个关键组件，提供高达 5GB 大小的快速、永久的文件存储。
- *Amazon* 弹性块存储（*EBS*）
 存储数据的关键模块。这是一个块级别的设备，可以附加到任何数据存储和检索的实例中。
- *AWS Import/Export*
 提供将大量数据导入或导出你的云解决方案的服务。
- *AWS* 高级支持（*Alexa Web Information Service*）
 支持所有 AWS 产品的服务，为 AWS 产品应用程序的构建和运行提供一对一的辅助。
- *Alexa web* 信息服务
 收集关于网站流量和结构信息的元数据的服务。
- *Alexa* 网站排名（*Alexa Top Sites*）
 基于流量和访问频率的网站排名服务。
- *Amazon* 土耳其机器人（*Amazon Mechanical Turk*）
 支持按需分配劳动力的一个协作解决方案。特别为将基于人的任务集成到计算系统中而设计，例如摄影、录音和其他数据提供者和消费者之间常见的以人为中心的任务。

现在你已经看到这样一个相关产品的列表，下面将集中讨论你在开始第一个云方案时需要知道的一些关键技术。这个列表虽然很短，跟我们刚刚列出来的长列表相比可能有点少，但这些是目前为止在云方案的构建中使用得最为频繁的技术。一旦掌握了这些，你就可以开始探索更多的高级服务了。

490

Amazon EC2

Amazon 弹性计算云（EC2）服务于 2006 年首次向测试用户发布，并在 2008 年公开。EC2 是 Amazon 云的动态计算能力背后的力量。它向动态分配的服务器实例提供虚拟硬

件，这些服务器实例支持主机上的操作系统和环境。EC2 是真正激发了云的东西。

EC2 的虚拟化使用开源的 Xen 技术，允许精细的硬件虚拟化和定制。Xen 虚拟化平台由 XenSource（最近被 Citrix 收购）创建，允许诸如 Linux、Windows 或 Solaris 客户操作系统在同一个机器上以虚拟机器的形式并发运行。

一个 EC2 上的虚拟机器被称为一个实例，你可以连接、监控和管理实例，就像运行在专用软件中的操作系统一样。

EC2 和 Xen 虚拟化的一个有趣的能力是支持 32 位和 64 位 CPU 的虚拟实例。一个 CPU 核或处理器内核被称为一个计算单元（CU，Computational Unit），它除了主管处理能力，还被 Amazon 用来作为成本乘数的单元。使用越多的计算单元，运行实例的成本就越大。因此，应该选择满足你的任务需要的最小计算单元。实例可能有多种类型，如表 14-1 所示。

表 14-1：实例类型

类型	CPU	内存	本地存储	平台	I/O	名称
小	1 个 EC2 CU	1.7 GB	160GB 实例存储（150GB 加上 10GB root 分区）	32 位	中	m1.small
大	4 个 EC2 CU（每个有 2 个虚拟内核，每个虚拟内核有 2 个 EC2 CU）	7.5 GB	850 GB 实例存储（2 × 420 GB 加上 10 GB root 分区）	64 位	高	m1.large
超大	8 个 EC2 CU（每个有 4 个虚拟内核，每个虚拟内核有 2 个 EC2 CU）	15 GB	1690 GB 实例存储（4 × 420 GB 加上 10 GB root 分区）	64 位	高	m1.xlarge
高 CPU，中	5 个 EC2 CU（每个有 2 个虚拟内核，每个虚拟内核有 2.5 个 EC2 CU）	1.7 GB	350 GB 实例存储（340 GB 加上 10 GB root 分区）	32 位	中	c1.medium
高 CPU，大	20 个 EC2 CU（每个有 8 个虚拟内核，每个虚拟内核有 2.5 个 EC2 CU）	7 GB	1690 GB 实例存储（4 × 420 GB 加上 10 GB root 分区）	64 位	高	c1.xlarge
高 CPU，超大	6.5 个 EC2 CU（每个有 2 个虚拟内核，每个虚拟内核有 3.25 个 EC2 CU）	17.1 GB	420 GB 实例存储（1 × 420 GB）	64 位	中	m2.xlarge

类型	CPU	内存	本地存储	平台	I/O	名称
高 CPU，双倍超大	13 EC2 CU（每个有 4 个虚拟内核，每个虚拟内核有 3.25 个 EC2 CU）	34.2 GB	850 GB实例存储（1 × 840 GB 加上 10 GB root 分区）	64 位	高	m2.2xlarge
高 CPU，四倍超大	26 EC2 CU（每个有 8 个虚拟内核，每个虚拟内核有 3.25 个 EC2 CU）	68.4 GB	1690 GB实例存储(2 × 840 GB 加上 10 GB root 分区）	64 位	高	m2.4xlarge

491 Amazon 基于虚拟机的计算时间收费，使用实例类型作为成本乘数。实例本身可以在多个区域（托管硬件的地方）中的其中一个上运行。关于实例的定价和配置的更多信息参见 *http://aws.amazon.com/ec2/#pricing*。

EC2 实例使用 AMI。AMI 由操作系统和预装载的附加软件组成。Amazon 将很多预创建的 AMI 分门别类，使得 EC2 的入门更加简单。例如，你可以在 Linux, Apache, MySQL, PHP/Perl/Python 软件堆栈上加载一个预创建的 AMI。

Amazon S3

同样在 2006 年，Amazon 创建了 S3 作为其首要的在线存储 Web 服务。S3 为开发者提供了简单、安全和永久的在线存储，提供了本质上无限容量的能力。从较高层次上说，S3 类似于（至少在概念上）一个存储区域网络（SAN），因为来自任何已连接设备的资源都是可用的。Amazon 根据存储数据和存取数据所消耗带宽的使用情况来收费。

与使用目录结构的传统文件系统不同，S3 使用一种称为桶（*bucket*）的对象存储机制，可以使用公共可读的名字来定义桶。也就是说，你可以在“mycompanyname”桶中存储任何东西，但注意使用一些常用的名字，例如“database”或“public”或“documents”等。大多数用户使用他们的域名来命名桶，但即使那样也无法阻止其他人在相同的桶中存储数据。

Amazon 把存储在桶中的东西称为对象（*object*），每个对象可以小到一个字节，大到 5GB。桶和其中的对象被存放在两个可用性区（*availability zone*）中的一个区内（一个区域包含北美的很多数据中心，另一个区域包括欧洲的数据中心），但是你能从任何地方访问它们。

而且，Amazon 提供允许大部分基于 Web 的应用使用 S3 的 Web 服务。

492 S3 机制并不意味着快速读 / 写机制，它最好的用途在于归档，像存储定制的 AMI 或者批量数据拷贝或备份。因此，你不希望用它来存储活动数据库中的数据。

Amazon EBS

2008 年 Amazon 发布了 EBS。这是向云计算前进的巨大一步。EBS 是一个虚拟块存储设备，就像磁盘一样。它不仅具备典型块设备的性能，提供快速读写能力，而且与运行实例无关。这真的很重要，因为过去用户不得不依赖于从 S3 或者云之外的设备中取数据，然后装载到实例上。但实例本身是易变的，所以当实例结束的时候（非预期或者无警告式的发生），你会失去该实例上的所有更改。所以没有 EBS 之前，你必须频繁地将应用程序备份到 S3 中，或者使用诸如卷管理器之类的工具。

有了 EBS 以后，现在用户可以创建独立的设备（称为卷或 EBS 存储），并将它们附加到任何正在运行的实例上，而不像 USB 硬件设备那样。EBS 卷对开发者来说是一种标准的块设备，其规模从 1GB 到 1TB 不等。EBS 卷跟 SAN 很相似，可以使用快照动态调整大小。这样很方便，可以使硬盘随着应用和数据的增长而增加。

也可以使用多个串列 EBS 卷以提高吞吐量和 I/O 性能。更好的是 EBS 卷在 Amazon EC2 的可用区域（available zone）之间复制，也就是说，即使你所在的区域发生灾难，你的数据仍然是可访问的。因此 EBS 比传统的磁盘存储系统更加可靠。可是，附加到 EC2 实例的 EBS 卷必须驻留在相同的可用区域内。

EBS 还支持使用 point-in-time 快照将数据备份到 S3 中。每次备份都是一个不同的快照：只有上一次快照之后发生变化的块才会被保存。point-in-time 为创建持久备份，特别是 MySQL 数据库的持久备份，提供了一种高效可靠的方法。

由于 EBS 卷拥有比常规磁盘更多的能力，所以 EBS 卷是存储 MySQL 数据库文件的完美解决方案。你可以得到类似的快照和备份功能，其可持续发展是无可比拟的。

它是如何工作的

在我们详细讨论设置 AWS 账户、运行 AMI 实例之前，要先从概念上理解如何与 Amazon Cloud 进行交互，以及虚拟服务器是如何实现的。图 14-2 描述了如何实现和执行 EC2 实例的概念模型。

Amazon Cloud工具

493

有两种工具可用于购买服务、初始化和管理资源：GUI 和命令行接口。实际上有两种类型的 GUI。Amazon 提供了一个基于 web 的控制台来访问其服务，还有很多 Web 浏览器的插件可供使用。Amazon 还提供了建立在 EC2 API 之上的一组实用工具（称为 EC2 命令行工具）。

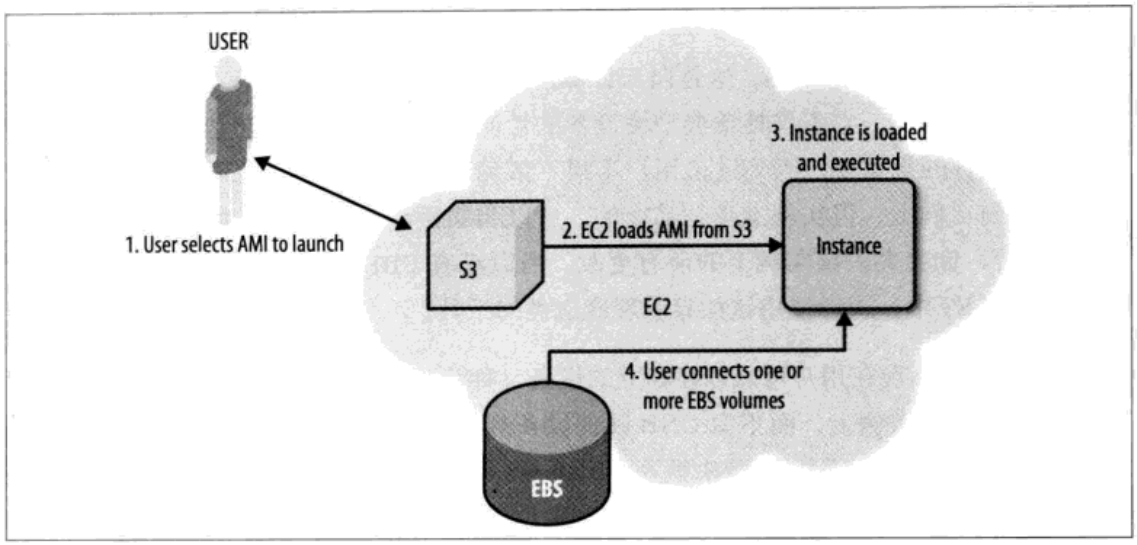


图14-2：AMI如何变为EC2实例

Amazon控制台

Amazon 提供了与其所有产品交互的基于 Web 的接口，称为 AWS 管理控制台。详见 *https://console.aws.amazon.com/ec2/home*。

首先需要一个 AWS 账号，接下来看看怎样启动 AWS 管理控制台。先介绍一下控制台，让大家对 AWS 和 EC2 的使用有更好的了解。

可以使用接口来创建实例、连接实例、创建 EBS 卷并将它们连接到实例，等等，这是使用 EC2 和其他 AWS 产品的默认机制。图 14-3 展示了典型用户的 AWS 管理控制台。

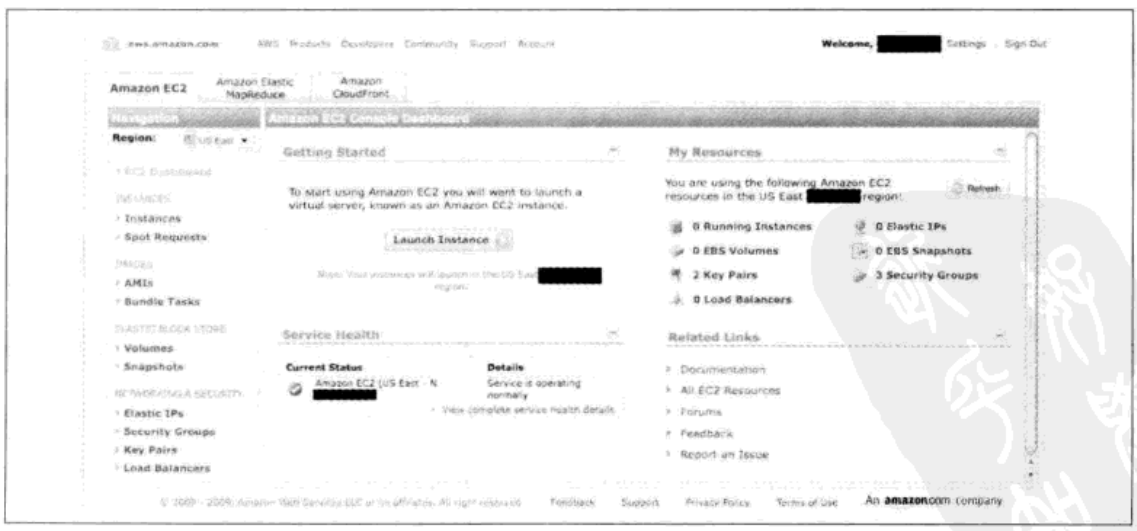


图14-3：AWS管理控制台



图 14-3 和本章的其他几处地方，我们已经屏蔽了一些特定的、与本书内容不相关的关于用户的上下文信息。

屏幕左边的中间位置是一个带有 Launch Instance 标签的按钮，这就是你开始创建 EC2 实例的地方。右边是 AWS 云资源的链接，如卷（EBS）、弹性 IP、负载均衡器等。 494

上方有三个标签页，用于访问不同的云服务组。默认标签页（如图 14-3 所示）是 Amazon EC2 页，还有 Amazon Elastic MapReduce 页——允许设置和执行网格计算解决方案，和 Amazon CloudFront 页——控制你的 Web 内容。

下一节将深入讨论创建 EC2 实例的细节和步骤。

浏览器插件

如果你想使单个网页更加强大，可以使用 Mozilla Firefox 的浏览器插件 Elasticfox。可以在 *http://developer.amazonweb services.com/connect/entry.jspa?externalID=609* 下载此插件。

Elasticfox 是一个基于 Web 的 GUI，执行完整的 EC2 工具 API，允许你控制 EC2 实例的方方面面，从创建实例到创建卷然后将它们连接到实例等。从很多方面看，它都比 AWS 管理控制台更强大，因为它使一切都在你的掌控之下。图 14-4 显示了 Elasticfox 的实例。

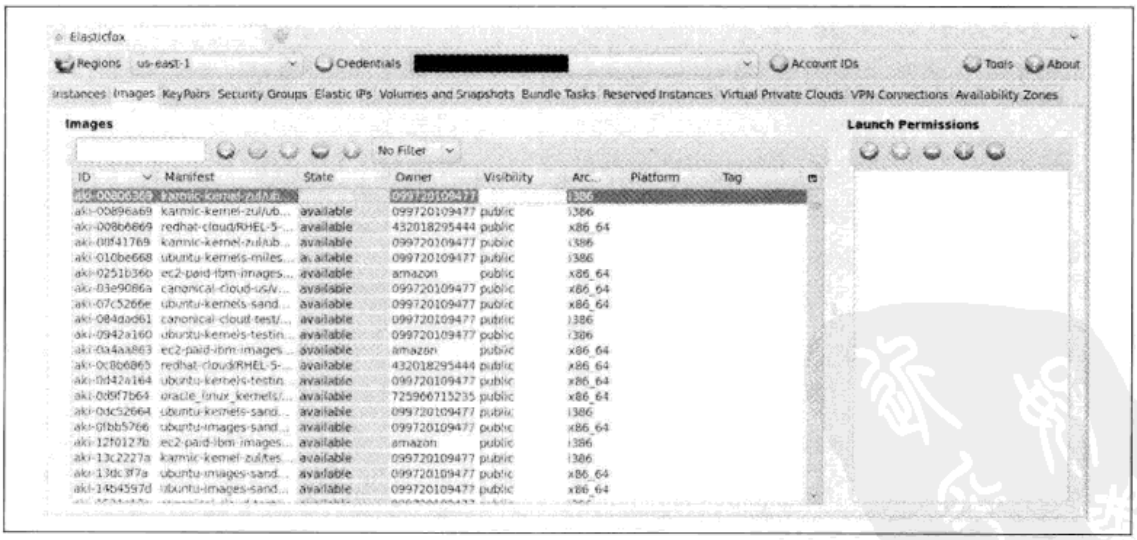


图14-4：Elasticfox Firefox插件

确保先到 *http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1797* 阅

读入门文档，然后再搭建和配置用于 EC2 访问的 Elasticfox。

495 其他一些供应商和开发者正在创建基于 web 的 AWS 管理控制台的替代品。其中最著名的是 shareVM 项目，可以在 *http://blog.sharevm.com/2009/01/09/web-based-ec2-console-alternative-to-elasticfox/.* 找到更多相关信息。

EC2命令行工具

还可以使用命令行工具与 EC2 进行交互。有两组命令行工具：API 和 AMI 工具。与 EC2 交互的 API 工具包括启动实例、创建和附加卷、管理安全组等工具。AMI 工具创建和管理 AMI。

EC2 API 工具。可以从 *http://developer.amazonwebservices.com/connect/entry.jspa?externalID=351&categoryID=88* 下载 EC2 API 命令行工具。按照下面的说明来安装和配置工具。

EC2 API 工具的文档包含在入门文档中，见 *http://docs.amazonwebservices.com/AWSEC2/latest/CommandLineReference/*。EC2 用户手册也有关于命令行工具的文档，用户手册可以在 *http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf* 下载。

下载和安装 EC2 API 工具要花一些时间，其中包括很多实用工具，使你可以在 EC2 中做很多事情。下面列出了更加常用的一些实用工具：

- `ec2-add-key-pair`
 创建一个新的 SSH 钥匙键对。

496
- `ec2-run-instances`
 启动 EC2 实例。必须至少声明映像名和钥匙键对，可以同时启动多个实例。
- `ec2-describe-images`
 列出所有可用的映像。输出结果包括映像 ID、映像在 S3 中的位置，以及映像是否可启动。可以使用很多参数来限制搜索结果。
- `ec2-stop-instances`
 停止或暂停实例。可以同时停止多个实例。
- `ec2-start-instances`
 启动或重置实例。可以同时启动多个实例。
- `ec2-terminate-instances`
 终止实例。可以同时终止多个实例。

只有几个常用的命令用于操作安全组、键、映像、卷等。详见用户手册。

EC2 AMI 工具。可以从 *http://developer.amazonwebservices.com/connect/entry.jspa?extern*

alID=368&categoryID=88 下载 EC2 AMI 命令行工具。并按照说明安装工具。

如果你想创建自己定制的映像，可能还要看看下面的文档：

http://docs.amazonwebservices.com/AWSEC2/latest/DeveloperGuide/
http://docs.amazonwebservices.com/AWSEC2/latest/CommandLineReference/

入门

Amazon AWS 产品是收费的解决方案，所以需要创建一个账户，为 Amazon 提供有效的付款来源。使用 Amazon Cloud 入门很简单。由于你需要为自己的学习曲线付费，如果进行小规模的练习（例如，使用小的实例，并且不会让它们运行很长时间），将只需花费一点就可以学会如何使用云。例如，本章的研究费用比一家流行快餐店的一顿饭还便宜。这种方式比买一台小型服务器便宜多了！



Amazon 的很多产品都是收费的，包括计算时间和存储量。确定当你的实例不再使用的时候终止实例，并清除临时存储。尽管每小时或单位数据规模的成本很小，你还是要付费的，即使你没有主动使用资源。这就像如果你出去度假两个星期却没有关灯，就不该为收到电费账单而感到惊讶。

下一节将展示如何获取 AWS 账户、启动实例、创建磁盘卷，并将它连接到你的运行实例上。本章的后面部分还会让你看看在云中使用 MySQL 是如此简单。 497

获取账户

首先需要一个 Amazon AWS 的账户。为了使用基本的云服务，必须创建账户，注册 EC2、S3 和 EBS 服务。幸好这个过程很简单：

1. 进入 AWS 网站。
2. 单击“Create an AWS Account”链接。
3. 单击“Sign Up Now”按钮。
4. 输入你的注册 ID（如 E-mail 地址），然后选择“I am a new user”按钮。单击“Sigin in using our secure server”按钮。



如果你已经有一个商业站点的 Amazon 账户，可以使用这个账户，而不用重新创建新的账户。

5. 选择密码，输入两次以验证密码。这里还要提供E-mail地址，单击“Create account”按钮。

6. 输入账户的完整详细信息，包括账单地址和联系方式。还可以选择性地使用自己的网站名和URL。你必须同意本页底部的AWS客户协议，然后继续。看完并同意协议后，单击“Agree”单选按钮，输入安全验证码，然后单击“Continue”按钮。

7. 到邮箱中查收确认邮件，按照邮件中的要求去做。

8. 进入你的账户，单击主页右边的Payment Method链接，进入Account页面提供账单信息。



直到你激活了付款选项才可以使用AWS EC2。

注册Amazon EC2（或任何其他服务或产品）：进入Products页面，单击蓝框中的链接，然后单击右上方的“Singnupfor”按钮，按照屏幕上的说明进行。注册S3和任何其他的服务或产品都是同样的步骤。

498

获取资格证书

访问EC2和其他AWS产品需要某种形式的安全协议。包括AWS注册名和密码、访问AWSAPI的访问密钥、访问AWS API的SOAP协议所需的X.509认证，以及用于访问EC2和CloudFront的密钥键对。要检查产品的注册需求，保证你拥有正确的资格证书。

Amazon推荐创建一个文件夹来存储你的私有密钥。创建这个文件夹要花些时间，下载私有密钥和其他安全资格之前要防止它被偶然发现。

Amazon登录和密码

为了使用AWS管理控制台和所有的账户行为，需要用AWS注册时的用户名和密码登录。

访问键ID和私密访问键

要使用查询API（如图片搜索）和很多GUI工具（如Elasticfox），需要一个访问键ID和一个私密的访问密钥。这些是最常用的请求资格证书。使用下面的步骤创建访问密钥：

1. 进入AWS主页面。

2. 单击“Account”选项卡。

3. 单击“Security Credentials”链接。



如果你尚未注册，则会要求你先注册。

4. 找到“Access Credemtials”部分，单击“Access Keys”选项卡。

5. 如果你在注册 AWS 时创建了访问键，它们就会被列出来，单击“Show”链接查看私密访问密钥。可以将这个密钥复制到你系统中的一个安全（访问保护的）文件中。

6. 如果没有访问密钥，或者你想创建一个新的访问密钥，可以单击“Create a new access key”按钮。

SOAP和EC2命令行工具

运行 EC2 命令行工具或者使用 SOAP 协议连接需要 X.509 认证和私有密钥。要定期轮换访问密钥以减少安全风险。Amazon 每 90 天会自动轮换访问密钥。使用下面的步骤来创建 X.509 认证：

1. 进入 AWS 主页面。

499

2. 单击“Account”选项卡。

3. 单击“Security Credentials”链接。



如果你尚未注册，则需要先注册。

4. 找到“Access Credemtials”部分，然后单击“X.509Cerificates”选项卡，就会显示你所拥有的所有认证，可以选择激活或停用。

5. 单击“Create a new certificate”链接，创建一个新的证书，且即将开始私有密钥的下载。



可以有两个证书，如果已经有两个证书了，需要删除一个才能继续创建新的证书。

6. 当出现对话框的时候，单击“Download Private Key File”按钮，然后将私有密钥保存到事先创建好的文件夹中。

7. 单击“Download X.509certificate”按钮，将认证保存到相同的文件夹下。

8. 下载完毕后单击“Close”按钮。

现在你已经有一个以 *pk-* 开头的文件和另一个以 *cert-* 开头的文件，它们就是你的私有密钥和认证文件。

CloudFront的资格证书

CloudFront 需要通过另一种特殊的密钥对机制进行访问。简单地称其为密钥键对通常会与实例中使用的密钥键对混淆。CloudFront 密钥键对与 X.509 认证工作原理类似，因为你必须在账户的资格证书页面才能访问，并且每次只有两个可用的密钥键对。使用下面的步骤来创建 CloudFront 密钥键对：

1. 进入 AWS 主页面（*http://aws.amazon.com/*）。

2. 单击“Account”选项卡。

3. 单击“Security Credentials”链接。



如果你尚未注册，则需要先注册。

4. 找到“Access Credemtials”部分然后单击“Key Pairs”选项卡，将显示你所拥有的所有证书，可以选择激活或停用。

5. 单击“Create a new key pair”按钮创建一个新的密钥键对，然后即将开始私有密钥键的下载。



可以有两个密钥键对，如果已经有两个了，则需要删除一个才能再创建新的。

实例的资格证书

实例需要通过 SSH 密钥键对来访问。SSH 密钥键对在 AWS 管理控制台内创建，当然也可以在需要的时候创建，最好是每个实例创建一个以减少事故或故意泄露。



至少需要创建一个 SSH 密钥键对才能启动第一个实例。

有了这些密钥就可以访问正在运行的实例，而不用记住和管理密码。在实例中嵌入其中一个密钥，并需要提供你注册时的另一个密钥来进行认证过程，可以给这个密钥键对命名以便于管理。启动映像时必须要声明密钥键对名。

下面的过程将帮你创建新的 SSH 密钥键对：

1. 进入 AWS 管理控制台。



如果你尚未注册，则需要先注册。

2. 单击页面左边的“Key Pairs”链接。
3. 单击“Create Key Pair”按钮，输入一个唯一的名字。
4. 单击“Create”按钮，就会创建密钥键对，并开始下载私有密钥。将私有密钥保存到事先创建好的文件夹中。
5. 单击“Close”按钮。

其他资格证书

账户安全资格证书页面的底部是你的注册资格证书和账户标识符，如果你想更换邮件地址或修改密码，可以在这个页面进行。如果你想和其他 AWS 账户协作，需要将账户标识符提供给这些账户的持有者，同样他们也需要将自己的账户标识符提供给你。你的账户标识符在这个页面的底部。 501

你还可以注册一个称为 AWS 多因素认证的可选服务，这是一个使用密钥生成器来进一步增加账户安全性的认证方式。如果 AWS 的安全性很重要，就要考虑多了解一下 AWS 多因素认证可以提供哪些服务，这个页面上的按钮和链接信息可以帮助你决定是否应该使用它。

使用AWS管理控制台运行实例

在 EC2 中运行实例很简单。最初可能并不那么容易，好在 Amazon 做了大量的工作使得初始学习曲线尽可能和缓。

要想在 EC2 上运行实例，先进入 *http://aws.amazon.com* 页面，然后单击“Sign in to the AWS Nlanagement Consde”链接，就会在屏幕中间靠左看到“Launch In Stance”按钮（如图 14-5 所示）。

接下来选择需要启动的映像。出现一个请求实例向导，第一页给出了 Amazon 提供的所有可用的映像列表。有多个选项卡，可以选择“Ouick Start”（Amazon 提供的预配置的常用任务映像），“My AMI”标签（如果你已经创建了自己的 AMI），还有“C ommunity AMI”标签（社区开发者们创建的可供你使用的 AMI）。

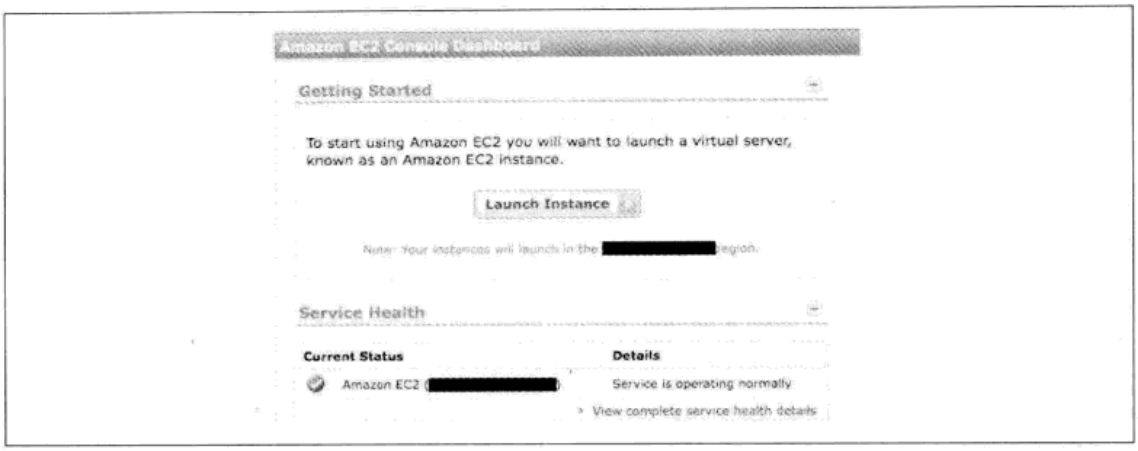


图14-5：在EC2中启动实例



有些社区 AMI 可能是收费的，特别是带有特有配置和专用软件的 AMI。可以使用免费 AMI，这样可以避免额外费用，或者可以在使用 AMI 之前先确定它的价格。

既然你在使用 MySQL，那么标记有 LAMP Web Starter 的映像就是个不错的选择，因为在云中使用 MySQL 需要了解很多入门知识，而 LAMP 堆栈是运行在 Fedora 主机上的。图 14-6 给出了选择映像的对话框。

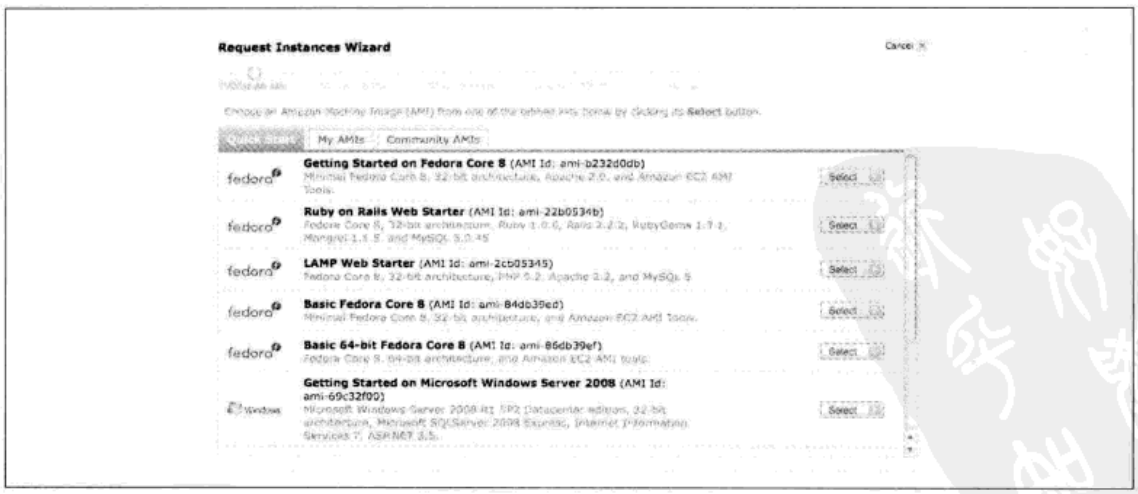


图14-6：选择要启动的映像

注意在“Quick Start”的列表中还有一个 Windows server 选项。到目前为止，大多数映像都是基于 Linux 的，但 Windows 开发者们也正在努力使 Windows 映像变得更加普遍。是否会有人创建出一个 Mac OS X 映像还有待观察（考虑到 Mac 平台的专有性，可能不会有这种可能）。至于 Windows，使用 Windows 映像需要提供你自己的 Windows 产品密钥。

确定要选择哪个映像之后，单击其名字右边的“Select”按钮，进入下一个页面（如图 14-7 所示）选择要启动的实例数目（可以一次启动多个实例）和实例类型（参见表 14-1）。大部分情况下，small 类型就足够了。



图14-7：实例的详细信息

还要选择可用区域。选择离你最近的可用区域。最后还必须选择是启动实例还是请求一个现场映像。

现场映像是一个按需请求的实例，它是按使用需求收费的（也就是按小时）。越多人请求实例，使用价格就会上升，请求较少价格就会下降。所以你可以选择对你有价值的实例，等到价格平稳下来并且当前收费降到阈值以下再行动。运行一些对时间要求不敏感的计算任务时，这是一个很好的低成本的办法（也就是说，你可以在给定的时间段内的任何时刻运行这些计算任务）。 503

如果你已经选择了实例数目、实例类型和可用区域，单击对话框底部的“Continue”按钮。



在AWS中创建的虚拟资源都有一个以类型简写形式开头的标识符，如实例以*i-*开头，卷以*vol-*开头，快照以*snap-*开头等。这样可以让你轻松地看到自己正在操作的是什么。

在如图14-8所示的下一个页面中，你要指定内核ID和RAM磁盘ID。大部分实例的启动使用默认设置。关于如何确定特定的内核和磁盘可以参见本页面的链接进一步学习。还可以选择打开云监控，并记录启动实例过程的备注。

504

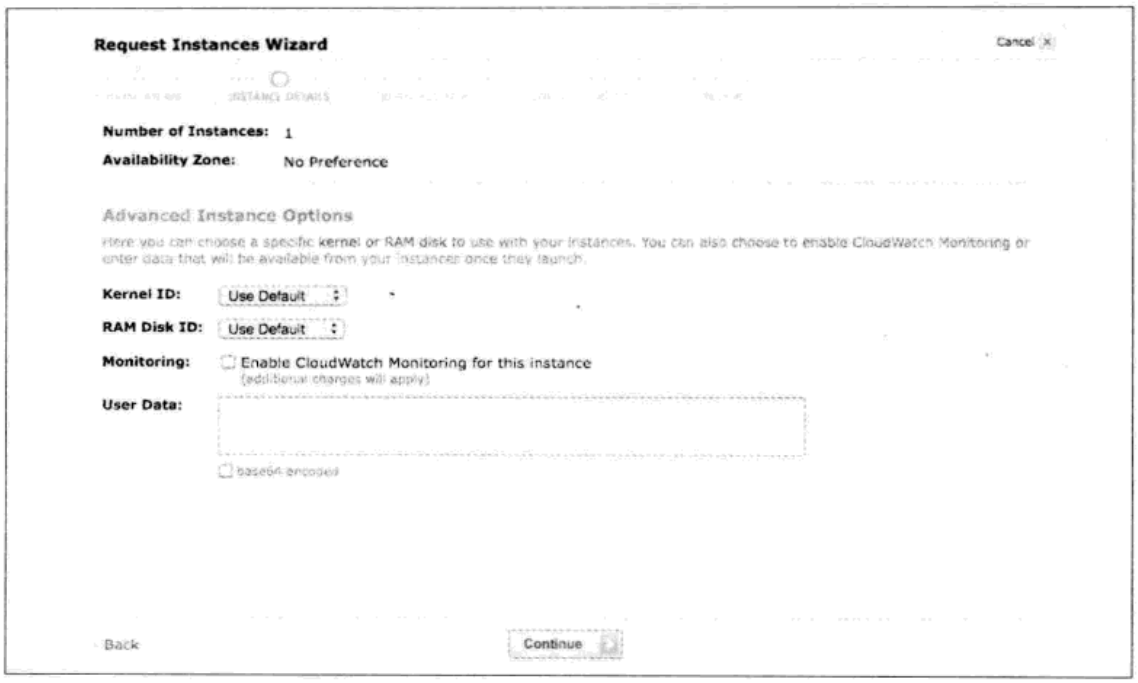


图14-8：高级实例选项

完成这些选择之后单击“Continue”按钮，进入图14-9所示的页面，指定访问这个实例所使用的SSH密钥键对。可以使用已有的SSH密钥键对，或创建一个新的密钥键对，或不用任何密钥键对直接继续（不推荐）。单击“Create a new Key pair”按钮来创建一个新的钥匙键对并将其命名。完成以后单击“Continue”按钮。

505

下一步，必须选择安全组或创建一个新的安全组，为你的实例建立防火墙设置。单击“Create a new security group”单选按钮，输入安全组的名字及其描述（保证该描述有意义，如图14-10所示）。

然后，定义允许的连接类型。默认情况下，这个对话框包括SSH的TCP端口22（需要使用EC2 SSH钥匙键对进行连接）和任何其他与映像相关的访问。例如，LAMP映像包括HTTP的TCP端口80和MySQL的TCP端口3306。图14-10给出了LAMP映像的

默认安全设置的例子。

完成这些选择之后，单击“Continue”按钮。

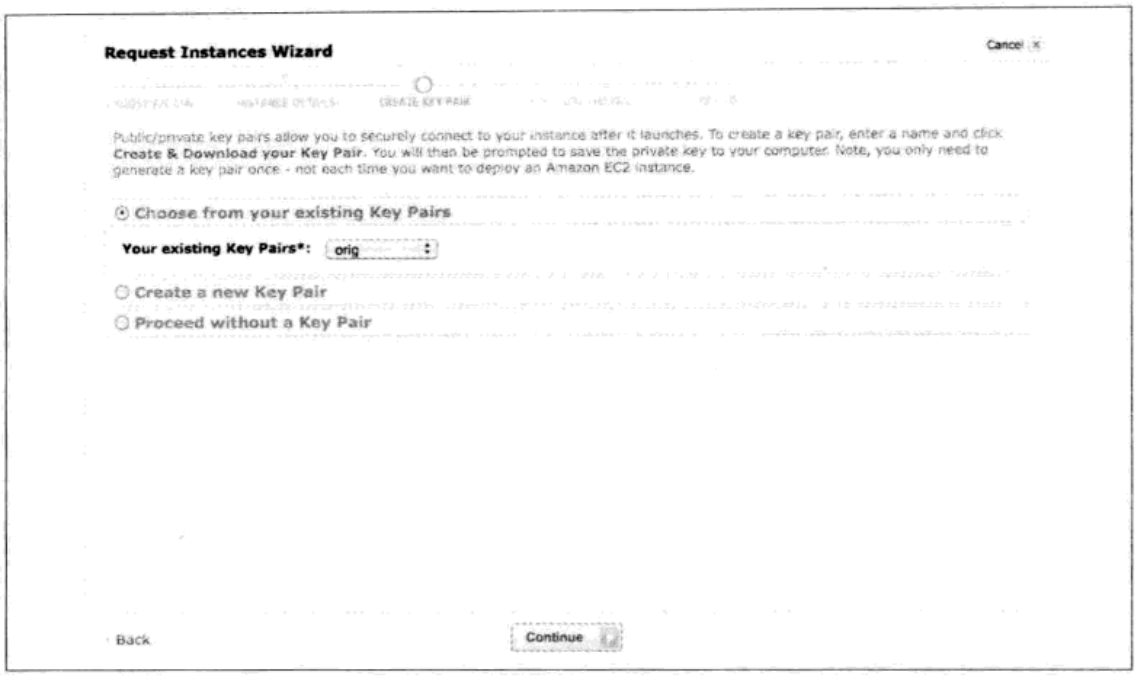


图14-9：创建键对

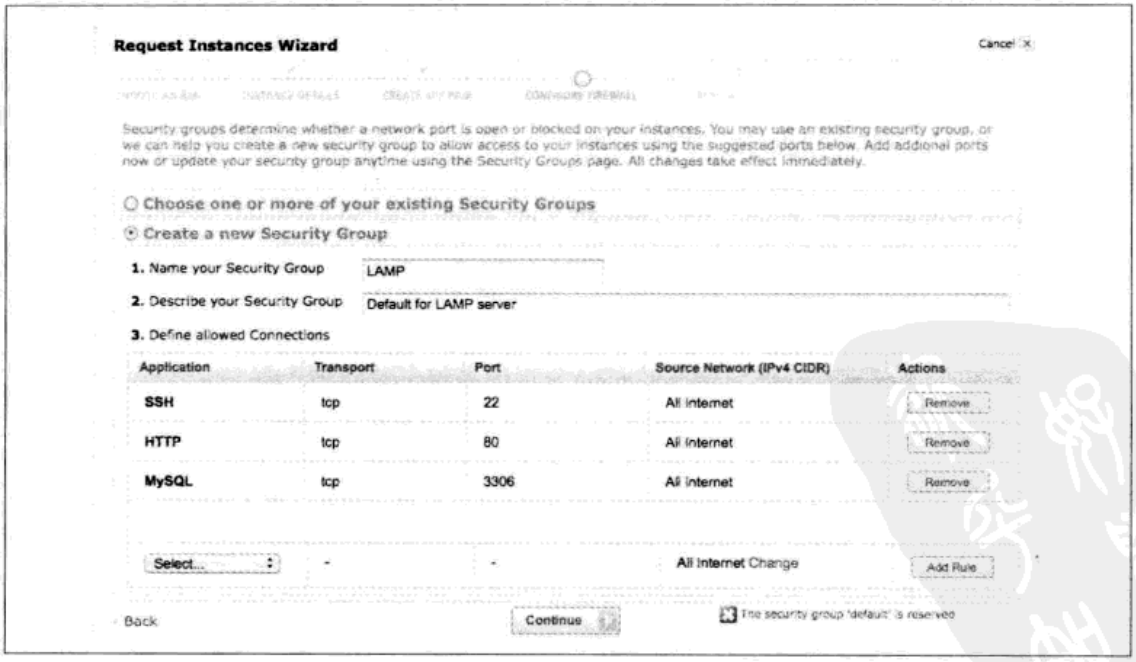


图14-10：设置网络访问控制

进入下一页，如图 14-11 所示，再次确认你的选择，并做必要的改动。准备好后单击“Launch”按钮启动实例。

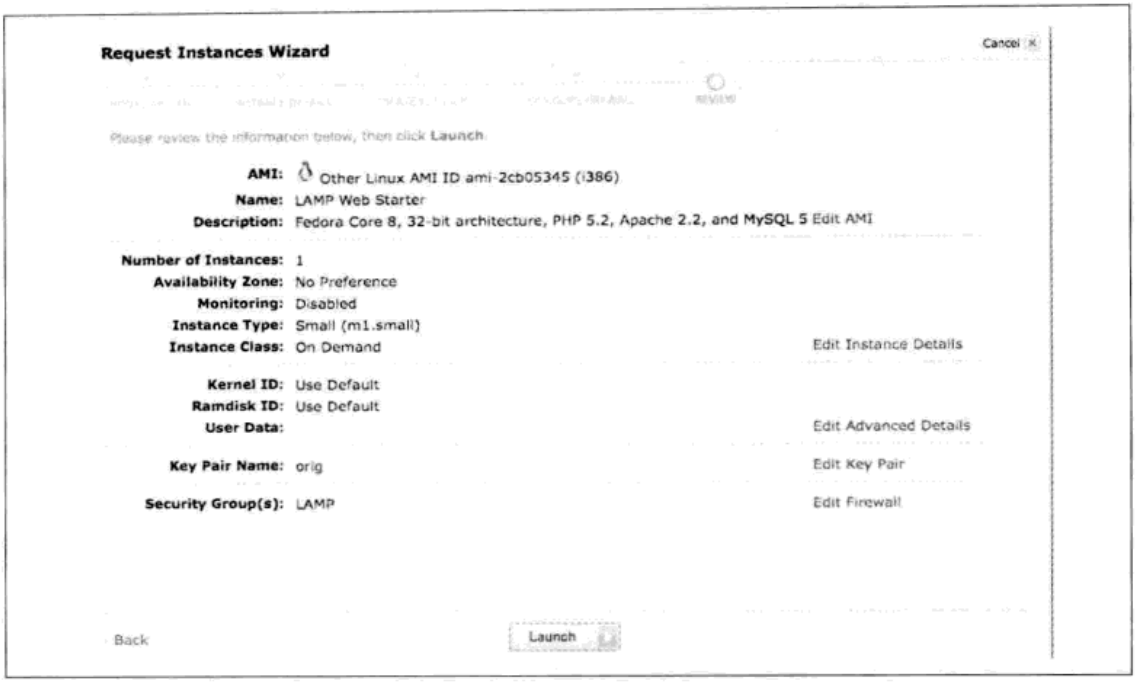


图14-11：最后检查启动的实例



也可以单击“Back”按钮返回到前面的对话框。

506 接着会出现一个对话框告诉你实例的状态信息，实例（或者多个实例）会在 AWS 管理控制台的“My Instances”部分显示。返回控制台，单击“My Resources”部分的“running instances”链接，将显示正在启动的不同阶段的实例。启动完成后，可在列表中查看其详细信息。

图 14-12 显示了正在运行的映像信息，可以看到连接信息及映像的描述信息、AMI ID、状态和实例类型。

可以在这个页面上执行不同的实例操作：启动和停止（暂停）实例、终止实例、甚至连接实例。在执行这些操作之前，首先必须选中一个或多个实例旁边的复选框。

现在让我们来连接实例。检查实例旁边的方框，然后在“Instance Actions”下拉列表中选择“connect”选项，如图 14-13 所示。这个对话框给出了连接实例所需的命令。打开终端或 command shell，输入这些命令。单击“close”按钮退出对话框。

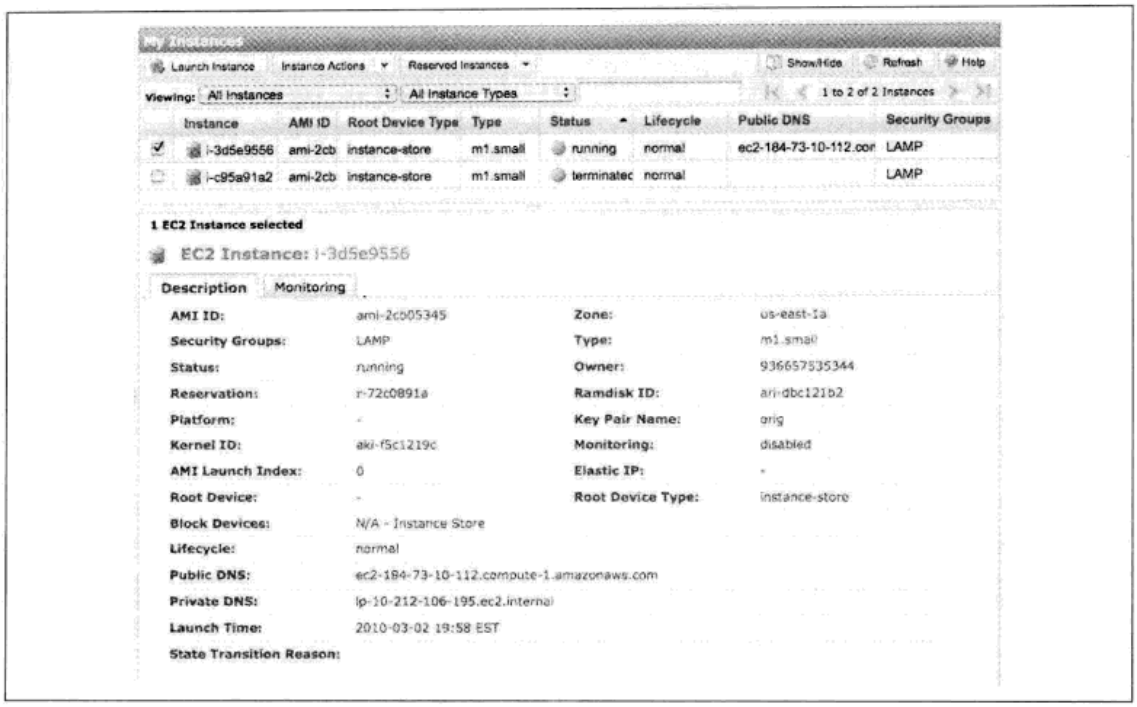


图14-12：我的实例

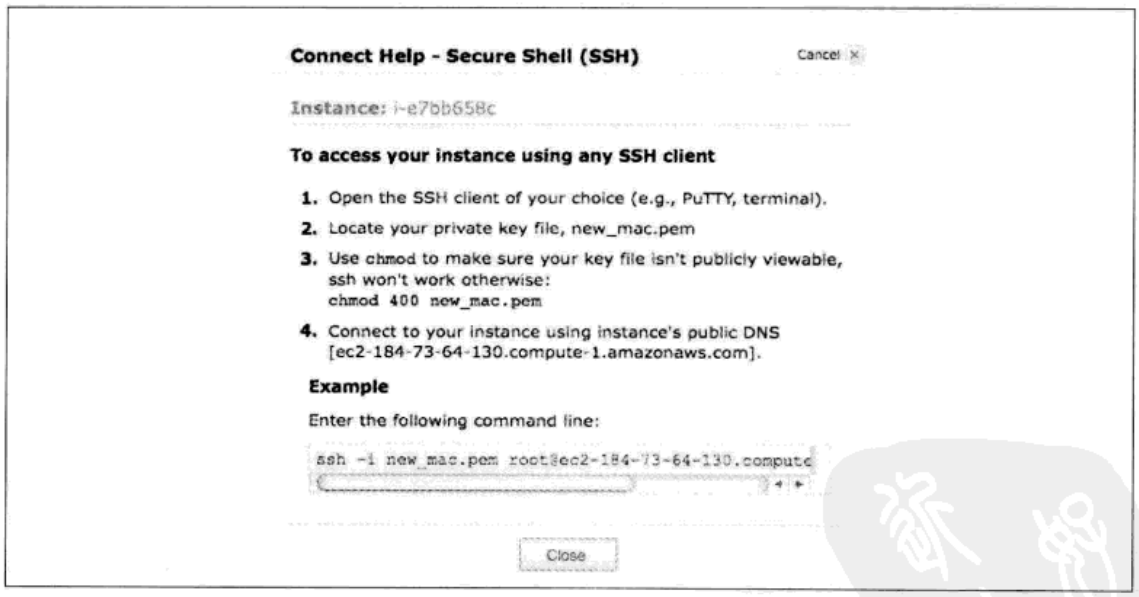


图14-13：连接帮助对话框



适当确保 SSH 密钥的安全性。例如，使用 CHMOD 0400 <key>命令保证密钥的安全性。

507 连接实例以后，你将看到如下的终端会话（SSH）：

```
Chucks-MacBook-Pro:~ Chuck$ ssh -i ./ec2_credentials/new_mac.pem
root@ec2-184-73-64-130.compute-1.amazonaws.com

       __|  __|_  )  Fedora 8
       _|  (     /    32-bit
      ___|\___|___|

 Welcome to an EC2 Public Image
                      :-)

    Base

 --[ see /etc/ec2/release-notes ]--

[root@ip-10-245-114-64 ~]#
```

有了root访问权限，你就可以在实例上执行任何操作。实例上的实验结束后，返回
508 AWS管理控制台的“My Instances”页面，检查实例旁边的方框，在“Instance Actions”下拉列表中选择“Terminate”选项，从而终止该实例。需要确认该操作以继续后面的操作。

有很多可用的实例操作，如查看系统日志、启动更多的实例、重启实例、停止或启动实例、启用和禁用监控等。

实例被终止以后，会显示终止状态。退出AWS管理控制台之前，最好确保已经终止了所有你不想继续运行的实例，因为其他的活动状态可能并不会停止计费周期，否则，你可能会对执行时间的账单感到惊讶。

使用EC2 API工具启动实例

现在你已经看到从AWS管理控制台中启动实例是多么简单，下面来看看如何使用EC2 API工具启动实例。

一旦安装好所需的工具以后，你就可以跟着我们一起启动和连接映像了。

509 第一步是使用`ec2-describe-images`命令列出所有的可用映像。下面的例子中使用grep命令搜索支持MySQL的映像。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-images -o self -o amazon | grep mysql
IMAGE    ami-225fba4b    ec2-public-images/fedora-core4-apache-
    mysql-v1.07.manifest.xml
    amazon    available    public      i386    machine         instance-store
IMAGE    ami-25b6534c    ec2-public-images/fedora-core4-apache-mysql.
    manifest.xmamazon
    available    public        i386    machine              instance-store
```

```
IMAGE    ami-255fba4c    ec2-public-images/fedora-core4-mysql-v1.07.
    manifest.xmlamazon
    available    public        i386    machine              instance-store
IMAGE    ami-22b6534b    ec2-public-images/fedora-core4-mysql.
    manifest.xml
    available    public        i386    machine              instance-storee
```



你也可以用 Cygwin 在 Microsoft Windows 上使用这些命令。关于 Windows 上工具的详细设置参见 Amazon EC2 用户手册。

列出所有可用映像后，选择 LAMP 映像并使用 `ec2-run-instances` 命令启动。此命令需要提供以下参数：映像名（AMI ID）和已下载的 SSH 密钥文件。下面给出了该命令的返回信息：实例 ID、AMI ID、状态（pending）、所使用的密钥、实例类型、时间、区域、是否启用监控和存储类型：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-run-instances ami-225fba4b -k new_mac
RESERVATION    r-2249194a    936657535344    default
INSTANCE        i-75af711e    ami-225fba4b           pending    new_macm1.
small
    2010-03-09T02:13:27+0000    us-east-1d              monitoring-disabled
    instance-store
```

使用 `ec2-describe-instances` 命令检查实例的状态。该命令返回的信息与前面相同，但返回的是最新的状态信息。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-instances
RESERVATION    r-2249194a    936657535344    default
INSTANCE        i-75af711e    ami-225fba4b          pending    new_macm1.
small
    2010-03-09T02:13:27+0000    us-east-1d             monitoring-disabled
    instance-store
```

一旦实例正在运行，这个命令就会返回如下的信息。注意这里的实例 IP 地址。你会看见两个 IP 地址：第一个是公共 IP 地址，可以用这个 IP 地址（或实例名）登录；第二个是私有 IP 地址，仅用于云内部的实例之间的通信。 510

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-instances
RESERVATION    r-2249194a    936657535344    default
INSTANCE        i-75af711e    ami-225fba4b    ec2-184-73-9-65.compute-1.
    amazonaws.com
    domU-12-31-39-02-EC-E7.compute-1.internal    running    new_mac    0
    m1.small
    2010-03-09T02:13:27+0000    us-east-1d              monitoring-disabled
    184.73.9.65    10.248.243.21          instance-store
```

为了访问实例，你需要授权给安全组。这里我们给出使用 ec2-authorize 命令设置默认安全组的捷径，并允许 SSH 的 TCP 端口为 22，如下面的例子。关于该命令的完整描述及使用 EC2 API 工具设置自定义安全组的例子，详见 Amazon EC2 用户手册。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-authorize default -p 22
GROUP        default
PERMISSION        default    ALLOWS    tcp    22    22    FROM    CIDR
    0.0.0.0/0
```

现在我们使用 SSH 连接实例。

```
Chucks-MacBook-Pro:~ Chuck$ ssh -i ./ec2_credentials/new_mac.pem
root@ec2-184-73-9-65.compute-1.amazonaws.com

      __|  __|_  )  Rev: 2
      _|  (     /
     ___|\___|___|

 Welcome to an EC2 Public Image
                        :-)

    Apache2+MySQL4


    __ c __ /etc/ec2/release-notes.txt

[root@domU-12-31-39-02-EC-E7 ~]# exit
logout


connection to ec2-184-73-9-65.compute-1.amazonaws.com closed.
```

成功！最后，使用 ec2-terminate-instances 命令退出实例并关闭实例，此命名需要实例 ID 作为参数（实例 ID 可从 describe-instances 命令的返回结果中获得）。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-terminate-instances i-75af711e
INSTANCE    i-75af711e    running    shutting-down
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-instances
RESERVATION    r-2249194a    936657535344    default
INSTANCE    i-75af711e    ami-225fba4b    ec2-184-73-9-65.compute-1.
amazonaws.com
    domU-12-31-39-02-EC-E7.compute-1.internal  shutting-down  new_macm1.small
    2010-03-09T02:13:27+0000    us-east-1d              monitoring-disabled
    184.73.9.65    10.248.243.21           instance-store
```

511 现在你的实例即将终止。再次（或多次）运行 describe-instances 命令，以验证实例是否已经终止。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-instances
RESERVATION     r-2249194a      936657535344      default
INSTANCE     i-75af711e     ami-225fba4b            terminated     new_mac     0
        m1.small     2010-03-09T02:13:27+0000     us-east-1d
        monitoring-disabled                  instance-store
```

如你所见，在 AWS 管理控制台中启动实例的所有功能都可以通过 EC2 API 工具实现。比起 GUI 环境，很多人更喜欢以这种方式使用 EC2。当然，你也可以选择对你来说最合适的方式。

使用磁盘

现在已经可以随意启动实例，该是看看磁盘存储的时候了。本节将讨论实例内部的磁盘存储（称为实例存储或实例卷）是怎样工作的，如何备份这些数据，以及如何通过创建 EBS 卷并将其连接到正在运行的实例来使用永久存储。我们用一个例子总结了怎样对 EBS 卷进行快照备份。



在创建 EBS 卷的时候，你可能想终止正在运行的实例，但该创建过程快而简单，所以如果让实例继续运行并不会花太多使用量。

使用实例存储

每个实例都有固定数量的磁盘存储供你免费使用。然而，必须将数据复制到一个永久存储区域，因为实例关闭时数据会丢失（而且任何系统改变也会丢失）。

对于类似 MySQL 这样的数据库系统这一点尤其重要。不要将数据存在实例存储上，因为如果实例因为某种原因终止了，数据就会丢失。

也可以对数据做定期备份，或将一个永久存储区域附加到你的实例上，而完美的解决方案就是前面所述的 EBS。有了 EBS，你可以创建一个卷来保存数据，实例终止后存储依然存在。

Amazon EC2 用户手册中有关于实例存储的其他信息，包括 RAID 配置和数据备份。

通过AWS管理控制台使用EBS卷

512

首先创建卷，然后启动一个实例，并连接这个实例。用下面的步骤在 AWS 管理控制台中创建新的 EBS 卷：

1. 进入 AWS 管理控制台。

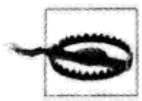


如果尚未注册，则需要先注册。

2. 单击页面左边的“Volumes”链接。

3. 单击“Create Volume”按钮，给出卷大小（单位为 GB）和可用区域，快照采用默认值。

4. 单击“Create”按钮，就会创建卷，并返回到 AWS 管理控制台。EBS 卷列表中显示卷状态信息。如果卷已经创建好，其状态就变为可用。



EBS 存储是收费的，包括与存储相关的 I/O 等，所以最好将所有已创建的当前不再使用的卷删掉。

要记住你是在哪个区域创建卷的，只能将卷连接到同一区域中运行的实例。图 14-14 显示了 EBS 卷创建好后的信息。



图14-14：EBS卷

注意在 EBS 卷的 Volume ID 框下面还有关于该卷的附加元数据的详细信息。

513 现在可以启动实例了。实例启动后，就可以将卷附加到实例上。返回到 EBS 卷列表，选择你想要附加到实例的卷，然后单击“Attach Volume”按钮，打开如图 14-15 所示的对话框，选择要连接的实例和磁盘设备（如 */dev/sdf*）。如果你使用的是 Windows，就会看到合适的盘符名称。

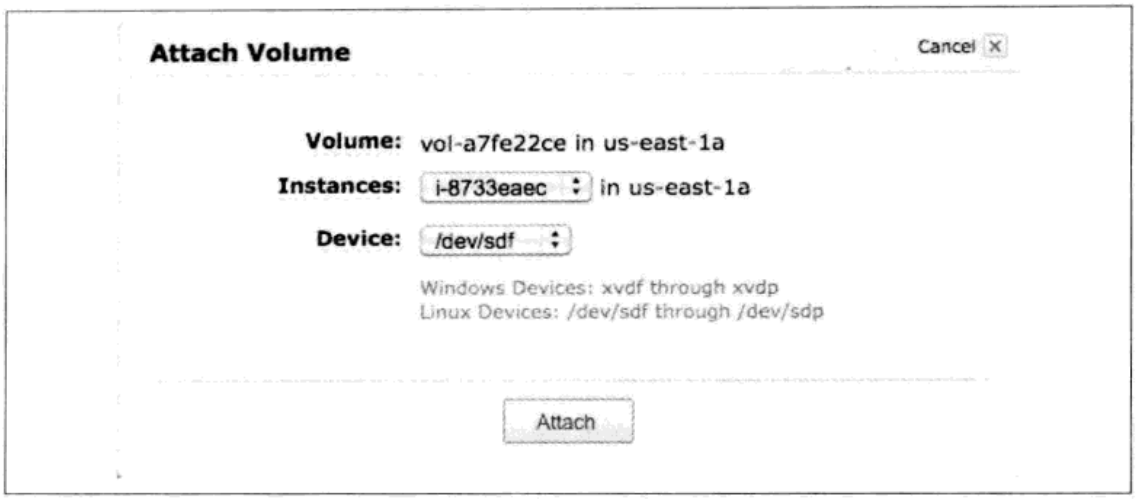


图14-15：选择需要附加的实例

现在你可以登录到实例并开始使用卷了。首先，将卷设为可用，下面给出典型的 Linux 例程。对 Windows 来说，使用 *diskmgmt.msc* 控制台格式化设备以供使用。

1. 创建一个 XFS 文件系统。

```
yes | mkfs -t xfs /dev/sdf
```

2. 创建装载位置。

```
mkdir /mnt/mysql-data
```

3. 装载设备。

```
mount /dev/sdf /mnt/mysql-data
```



如果你使用的映像本身没有 XFS，可能需要安装 AXF 实用工具（如 *xfsprogs*）。

要将卷断开连接，首先需要卸载设备，然后返回到 AWS 管理控制台，选择卷，然后单击“Detach Volume”按钮，然后进行确认操作。

通过AWS管理控制台使用EBS快照

514

如果用 EBS 卷存储数据，使用 EBS 快照可以在任何时间点进行快速备份。这会新创建该卷的一份拷贝，直至备份时间点的所有改变都被记录下来。

按照下面的步骤创建已有卷的快照：

1. 在数据库服务器上发出 FLUSH TABLES WITH READ LOCK 命令，保证所有数据都被写入磁盘，保证一致性。

2. 进入 AWS 管理控制台（*https://console.aws.amazon.com/ec2/home*）。



如果尚未注册则，需要先注册。

3. 单击页面左边的“Smapshots”链接。

4. 单击“Create Snapshot”按钮，选择要创建快照的卷，然后输入快照名。图 14-16 显示的是创建快照的对话框。

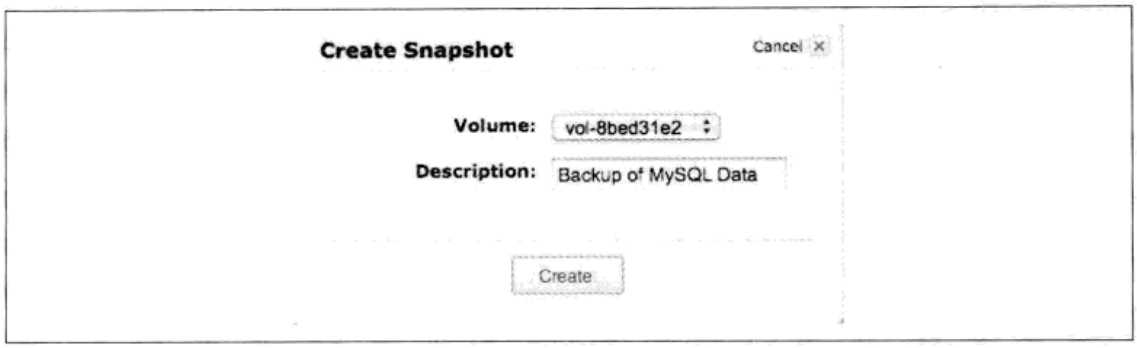


图14-16：创建快照对话框

5. 单击“Create”按钮，就会创建快照，返回 AWS 管理控制台。在 EBS 快照列表中列出了快照的状态。一旦卷是可用的，此状态就更改为可用。图 14-17 给出了 EBS 快照列表的例子。

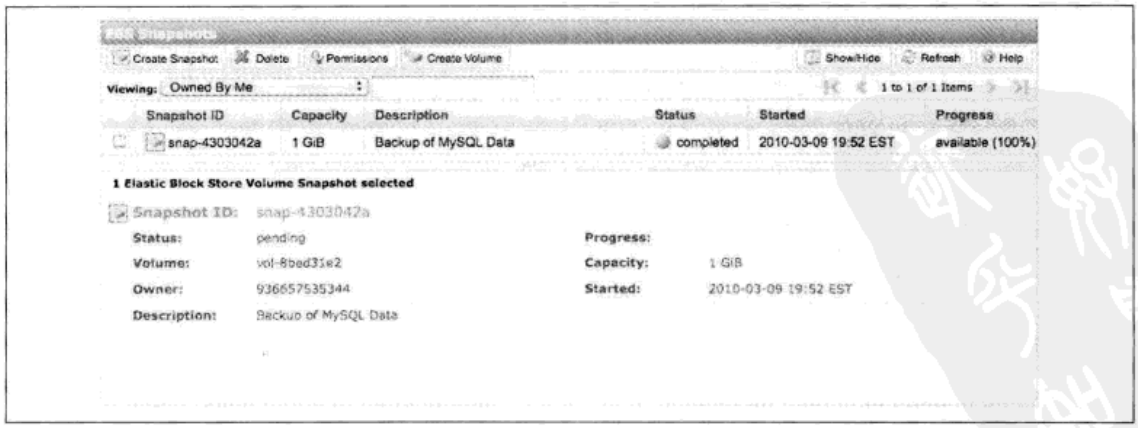


图14-17：EBS快照

6. 在数据库服务器上发出 UNLOCK TABLES 命令以解锁表。

如果要删除快照，在 EBS 快照列表中选择该快照，然后单击“Delete”按钮。该操作需要确认。

通过EC2 API工具使用EBS卷

还可以使用 EC2 API 工具来创建、附加、分离和删除卷。使用 ec2-create-volume 命令来创建卷，如下所示，至少要包含卷大小（单位为 GB）和区域参数：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-create-volume -s 1 -z us-east-1a
VOLUME    vol-7fec3016    1        us-east-1a    creating
    2010-03-10T00:28:09+0000
```

使用 ec2-describe-volumes 命令列出所有卷，其返回的信息与 AWS 管理控制台下的操作相同，下面给出了该命令的示例。请注意卷 ID，你可能需要它来执行其他卷操作。

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-volumes
VOLUME    vol-7fec3016    1        us-east-1a    available
    2010-03-10T00:28:09+0000
```

使用 ec2-attach-volume 命令将卷附加到正在运行的实例上，并给出卷 ID、实例 ID 和设备名，如下：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-attach-volume vol-7fec3016 -i i-414f962a
 -d /dev/sdf
ATTACHMENT    vol-7fec3016    i-414f962a    /dev/sdf    attaching
    2010-03-10T00:31:06+0000
```

根据前面的“通过 AWS 管理控制台使用 EBS 快照”小节部分所述的步骤，挂载或格式化卷以供使用，也可以按照前面的步骤从实例中卸载卷。

使用 ec2-detach-volume 命令断开卷到实例的连接，并需要卷 ID 作为参数，如下面的例子：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-detach-volume vol-7fec3016
ATTACHMENT    vol-7fec3016    i-414f962a    /dev/sdf    detaching
    2010-03-10T00:31:06+0000
```

516

最后，使用 ec2-delete-volume 命令删除卷，也要给出卷 ID：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-delete-volume vol-7fec3016
VOLUME    vol-7fec3016
```

既然你已经了解如何在卷上执行一些基本操作，下面我们就来看看怎样创建已有卷的快照。

通过EC2 API工具创建EBS快照

使用 ec2-create-snapshot 命令加上参数卷 ID，就可以用 EC2 API 工具创建快照了。下面的例子展示了如何从已有卷创建快照：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-create-snapshot vol-a7fe22ce
SNAPSHOT    snap-ad5a5dc4    vol-a7fe22ce    pending
    2010-03-09T22:45:22+000936657535344    1
```

创建快照需要一些时间。使用 ec2-describe-snapshots 命令可以查看快照列表：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-describe-snapshots
SNAPSHOT    snap-ad5a5dc4    vol-a7fe22ce    completed
    2010-03-09T22:45:23+0000    100%    936657535344    1
```

最后，使用 ec2-delete-snapshot 命令删除快照，这个命令会永久删除快照，所以要谨慎使用：

```
Chucks-MacBook-Pro:~ Chuck$ ec2-delete-snapshot snap-ad5a5dc4
SNAPSHOT    snap-ad5a5dc4
```

检查快照的状态，确保它们都已被删除。如果都已删除，那么 ec2-describe-snapshots 命令将返回空列表。

接下来怎么做

Amazon 云产品都很复杂，会带来较高的学习难度。但幸运的是，很多资源在 AWS 站点上都可以找到。这里我们列出了几个访问频繁和必读的链接：

http://aws.amazon.com/documentation/ec2/

http://aws.amazon.com/ec2/

http://aws.amazon.com/autoscaling/

http://aws.amazon.com/s3/

http://aws.amazon.com/ebs/

517

云中的MySQL

如果你还在疑惑在云环境中使用 MySQL 是否有什么特别之处的话，那么你可以松口气了。在云中运行 MySQL 与在“真实”机器上运行 MySQL 并没有什么不同。你仍然可以运行复制、配置高可用性和横向扩展方案、以及使用同样的工具来监控和管理这些解决方案等。

在云中使用 MySQL 真正不同的地方在于其快速部署能力，无论何时需要都可以快速部

署成百上千的服务器。本节来看看前面章节中的那些解决方案在云中是如何实现的，首先以 Amazon 云中如何使用 MySQL 复制为例。

MySQL复制和EC2

本节将展示在 EC2 中使用 MySQL 复制是多么简单。这个例子使用 LAMP Web Starter 映像启动实例，并使用 EC2 命令行工具连接该实例。然后将这个实例当成 Master，启动一个本地 MySQL 实例作为 Slave。这个过程与在本地环境中配置复制相同，唯一的区别就是本例中的 Master 是运行在云中的。

第一步是启动并连接实例。下面的代码展示了连接到一个正在运行的 EC2 实例的步骤和结果：

```
Chucks-MacBook-Pro:~ Chuck$ ssh -i ./keys/orig.pem
root@ec2-184-73-10-112.compute-1.amazonaws.com
The authenticity of host 'ec2-184-73-10-112.compute-1.amazonaws.com
(184.73.10.112)' can't be established.
RSA key fingerprint is cd:79:eb:e5:e9:2e:d6:a2:9c:79:65:2a:27:c5:1b:ba.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-184-73-10-112.compute-1.amazonaws.com,
184.73.10.112' (RSA) to the list of known hosts.
Permission denied (publickey,gssapi-with-mic).

          __|  __|_  )  Fedora 8
          _|  (     /    32-bit
         ___|\___|___|

 Welcome to an EC2 Public Image
                       :-)

    Base

 --[ see /etc/ec2/release-notes ]--

[root@ip-10-212-106-195 ~]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.45 Source distribution
Type 'help;' or '\h' for help. Type '\c' to clear the buffer

mysql>
```

518

下一步是关闭远程实例，编辑 *my.cnf* 文件（位于 */etc/my.cnf* 目录）。至少进行以下设置：server-id = 1，log-bin = mysql-bin。重启 MySQL 服务器，检查是否可以连接：

```
mysql> SHOW MASTER STATUS;
```

```
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |       98 |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

然后，必须发出 GRANT 语句以允许 Slave 连接：

```
mysql> GRANT REPLICATION SLAVE ON *.* TO 'rpl'@'%' IDENTIFIED BY 'rpl';
```

下一步是配置 Slave，并使其连接到 EC2 中的 Master 上。使用 CHANGE MASTER 命令，并提供 EC2 实例地址、Master 的二进制日志信息和复制用户账户等参数，例如：

```
mysql> CHANGE MASTER TO
  MASTER_HOST='ec2-184-73-10-112.compute-1.amazonaws.com',
  MASTER_USER='rpl',
  MASTER_PASSWORD='rpl',
  MASTER_PORT=3306,
  MASTER_LOG_FILE='mysql-bin.000001',
  MASTER_LOG_POS=98;
Query OK, 1 row affected (0.00 sec)
mysql> START SLAVE;
Query OK, 1 row affected (0.00 sec)
```

使用 SHOW SLAVE STATUS 命令查看 Slave 的状态，从而检查复制的工作情况。结果应该显示一个活跃状态的 Slave，如下面的例子：

```
mysql> SHOW SLAVE STATUS \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: ec2-204-236-207-171.compute-1.amazonaws.com
                  Master_User: rpl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000001
          Read_Master_Log_Pos: 325
               Relay_Log_File: mysqld-relay-bin.000002
                Relay_Log_Pos: 470
        Relay_Master_Log_File: mysql-bin.000001
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
519 Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
```

```
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 325
              Relay_Log_Space: 626
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
1 row in set (0.00 sec)
```

现在我们要在 EC2 实例的 Master 上创建数据库，然后在本地 Slave 上检查这个数据库：

```
mysql> CREATE DATABASE amazon_ec2_test;
Query OK, 1 row affected (0.00 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| amazon_ec2_test    |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.02 sec)
```

返回到 Slave，可以看见数据库已经被创建：

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| amazon_ec2_test    |
| employees          |
| mysql              |
| sakila             |
| test               |
+--------------------+
```

```
6 rows in set (0.00 sec)
```

520 花点时间自己试试，你就会发现在 EC2 中使用 MySQL 如此简单。一旦完成了上述步骤，就可以从 EC2 Master 复制数据到本地 Slave 了。我们把反向的复制交给你自己去配置和评估。要确保防火墙和路由器允许 TCP 端口 3306 传入流量。



记住，你在 Amazon EC2 中的实例是公开的，所以要小心设置你的安全访问等级和密码。例如，不要忘了设置 MySQL 安装的 root 密码，并禁用远程登录，这样只有授权用户才能连接到你的服务器。同样，只开启云中通信所需的端口。

EC2中使用MySQL的最佳实践

前面提到，MySQL 在物理硬件上能做的，在云中几乎都能做到，当然，专用硬件集群方案和基于硬件的高可用性方案要除外。但是必须认识到硬件是可以被虚拟化的，所以即使今天你使用了特定的硬件方案，很有可能明天它就会变成云中的虚拟资源了。

在 Amazon 云中使用 MySQL 与在自己的硬件上使用稍有不同。如果使用的是安装和配置有 MySQL 的 AMI，就可以快速连接 EBS 卷上的数据，然后启动服务器。或者你也可以创建自己的 AMI，采用特定的 MySQL 配置，自动设置将 *datadir* 指向 EBS 卷所需的一些小设置。

那么，在云中使用 MySQL 的最佳实践是什么呢？下面列出了一些达到最佳性能的常用实践：

- 每个 EC2 实例仅运行一个 MySQL 服务器
 如果拥有专有的计算单元和内存资源，MySQL 会运行得更快。考虑到创建新实例较容易，成本也相对较低，所以在较小的实例中运行单个 MySQL 实例比在较大的实例中运行多个 MySQL 实例性能更好。记住，无论何时都可以添加你需要的足够数量的实例，这样可以随意扩大或缩减规模。
- 在高使用率下采用较大的实例类型
 对高交易量或频繁读写数据库的情况，可以考虑使用较大的实例类型。较大的实例类型不仅有用更多的计算单元和内存，而且还有更高的 I/O 吞吐能力。这样的话，多花一点钱也是值得的。

521
- InnoDB 表空间使用附加 EBS 卷
 数据库是典型的 I/O 独占应用，会受到 I/O 限制。用多个 EBS 卷来隔离数据库，以及多个 InnoDB 表空间（例如每个 EBS 卷一个 InnoDB 表空间）来达到更好的 I/O 性能。
- 为数据分区“热身”
 在 EC2 中使用磁盘 I/O 有一个缺点：在往新分区中初始写入时“第一次写”的性能

不高。为了避免这个不足，可以执行一种散单命令访问这些分区来给它们“热身”。例如，使用 Linux dd 命令向磁盘写入。如果这个缺点依然存在且不可避免，至少数据库的第一个合法写入不会受影响。

- 正确配置 MySQL

 不过在 EC2 中运行 MySQL 并不是那么容易。为了满足特定的应用需求，你不能忽略 MySQL 系统的配置和优化。这样就可以更好地使用 EC2 的资源。

- 不要忘了监控

 要监控云中运行的 MySQL 服务器，你可以选择本书中提到的方法。还可以在你的服务器上使用 Amazon 云实例监控工具。因为即使服务器是虚拟化的，也并不能使它们不受失控查询和类似性能问题的影响。

- 使用 MySQL 复制

 前面的章节中你已经看到 MySQL 复制对于横向扩展、均衡负载和高可用性来说是多么重要。云让事情变得更加简单，因为你能创建足够数量的 MySQL 实例来实现几乎所有的 MySQL 高可用性方案。

- 使用标准 AMI

 从头创建自己的 AMI 允许调整服务器以满足特定的平台需求。不幸的是，除非有虚拟环境下运行宿主操作系统（或者启动一个已知的稳定的 AMI，然后做微小改动）的丰富经验，否则构建这个自定义的 AMI 将会十分费时且容易出错。所以，在任何可能的时候，请使用已有的 Amazon AMI 或其他已知的稳定的社区 AMI。

- 使用良好的安全做法

 云与其他任何与 Internet 连接的设备没有什么区别。要为 MySQL 实现有完整文档记录的安全协议（例如，不要让你的 root 账户的密码为空）。还要限制对 EC2 实例的访问，从而只有授权用户（和系统如 replication slaves）可以访问你的虚拟系统。EC2 有一个事先配置好的防火墙来限制最大流量。有些 AMI 开启了某些端口，如 MySQL 客户端使用的 AMI 开启 3306 端口，但你必须保证需要用到哪个端口才开启它。使用自定义的安全组来管理一组 EC2 实例之间的常用规则。

- 使用 *noatime* 和 *nodiratime* 选项挂载分区 <522>

 使用这两个选项挂载分区可以获得 10% 的 I/O 性能提升。这是因为 Linux 不会在每个写读取后都进行一次写操作。noatime 是 nodiratime 的子集。

- 在 MySQL 上使用 EBS

 前面提到，EBS 是一个块存储设备，具有高性能、从实例故障中恢复的可持续性及弹性。你不仅可以为你的数据和日志文件获得持久存储，而且实例终止时还可以把故障转移到其他服务器上。

- 使用 S3 执行快照

 使用 EC2 对你的卷执行快照，然后将其存储到 S3 上，这是备份策略的一部分。快

照是一个高效可靠地创建备份的方法，还提供了数据损坏时的快速恢复机制。

- 使用负载均衡

 均衡负载有利于高负载应用或大量并发连接。如前面章节所述，可以在多个 MySQL Slave 之间使用 MySQL 复制来提高读性能，并使用切片（sharding）提高写性能。可以使用 Amazon 弹性负载均衡（Amazon Elastic Load Balancing）资源或运行自己的软件负载均衡服务器（如 HAProxy）。关于 HAProxy 的更多信息请见 *http://haproxy.1wt.eu/*。

开源云计算

如果你担心实验或进一步学习带来的成本费用，那就考虑开源方案吧。Ubuntu 服务器预装了最新的开源云系统，称为 Eucalyptus，与 Amazon EC2 很相似，可以让你免费学习云计算。例如，如果你想学习怎样配置一个内部实验或开发用的云方案，可以装载 Ubuntu 云服务器立即开始。下面为你展示了如何在 Ubuntu 云中使用命令行工具启动实例。注意这些命令与本章前面所讲的 Amazon 命令很相似：

```
cbell@ubuntu-cloud:~$ euca-describe-images
IMAGE    eri-099C1159    image-store-1266350672/ramdisk.manifest.xml    admin
    available    public    x86_64    ramdisk
IMAGE    emi-DED7106D    image-store-1266350672/image.manifest.xml    admin
    available    public    x86_64    machine
IMAGE    eki-F52D10EB    image-store-1266350672/kernel.manifest.xml    admin
    available    public    x86_64    kernel
523 cbell@ubuntu-cloud:~$ euca-describe-volumes
VOLUME    vol-330A04B9    10    cloud9    in-use    2010-02-
17T18:54:43.589Z
ATTACHMENT    vol-330A04B9    i-41EE0860    unknown,requested:/dev/sdb
    2010-02-18T17:44:11.561Z
VOLUME    vol-32DC04A3    10    cloud9    in-use    2010-02-
17T18:54:41.002Z
ATTACHMENT    vol-32DC04A3    i-4DDB09AC    unknown,requested:/dev/sdb
    2010-02-18T17:44:11.561Z
cbell@ubuntu-cloud:~$ euca-describe-instances
RESERVATION    r-5C060A63    admin    default
INSTANCE    i-41EE0860    emi-DED7106D    172.19.1.3    172.19.1.3
    running    mykey    0    c1.medium    2010-02-18T17:32:12.441Z
    cloud9 eki-F52D10EB    eri-099C1159
RESERVATION    r-4027075E    admin    default
INSTANCE    i-4DDB09AC    emi-DED7106D    172.19.1.2    172.19.1.2
    running    mykey    0    c1.medium    2010-02-18T17:31:55.128Z
    cloud9 eki-F52D10EB    eri-099C1159
```

要想学习更多关于 Ubuntu 云系统的知识，并安装自己的私有云计算方案，参看下面的链接：

http://www.ubuntu.com/cloud

https://help.ubuntu.com/community/UEC

安装一个 Ubuntu 云计算环境需要至少两个支持虚拟化的多核服务器。

小结

我们已经了解了云计算，也看到了 Amazon 云方案的威力。我们发现旧的技术以新的方式组合起来可以变得多么焕然一新。云计算使得传统的信息基础设施成为过去。有了云计算，你可以使你的基础设施随着业务增长而增长，而不用聘请技术人员队伍去管理它。

本章中定义了云计算，介绍了云计算涉及的一些架构，还简单了解了 Amazon 的 EC2、S3 和 EBS 产品。还给出了如何向 EC2 环境复制及如何复制到 EC2 环境中去的例子。

最后讨论了如何在云中使用 MySQL，好在云里面的任何操作都比物理硬件上更快，且资源和资金投入更少。你甚至可以配置自己的 Amazon 云方案，来实现工作负载的自动伸缩。谁不喜欢这样呢？

"Joel！"

524

Joel 不由自主地一次按了好几个键，扬声器发出一阵警戒音。他抬起头，看见 Summerson 在他办公室门口正微笑着看着他。

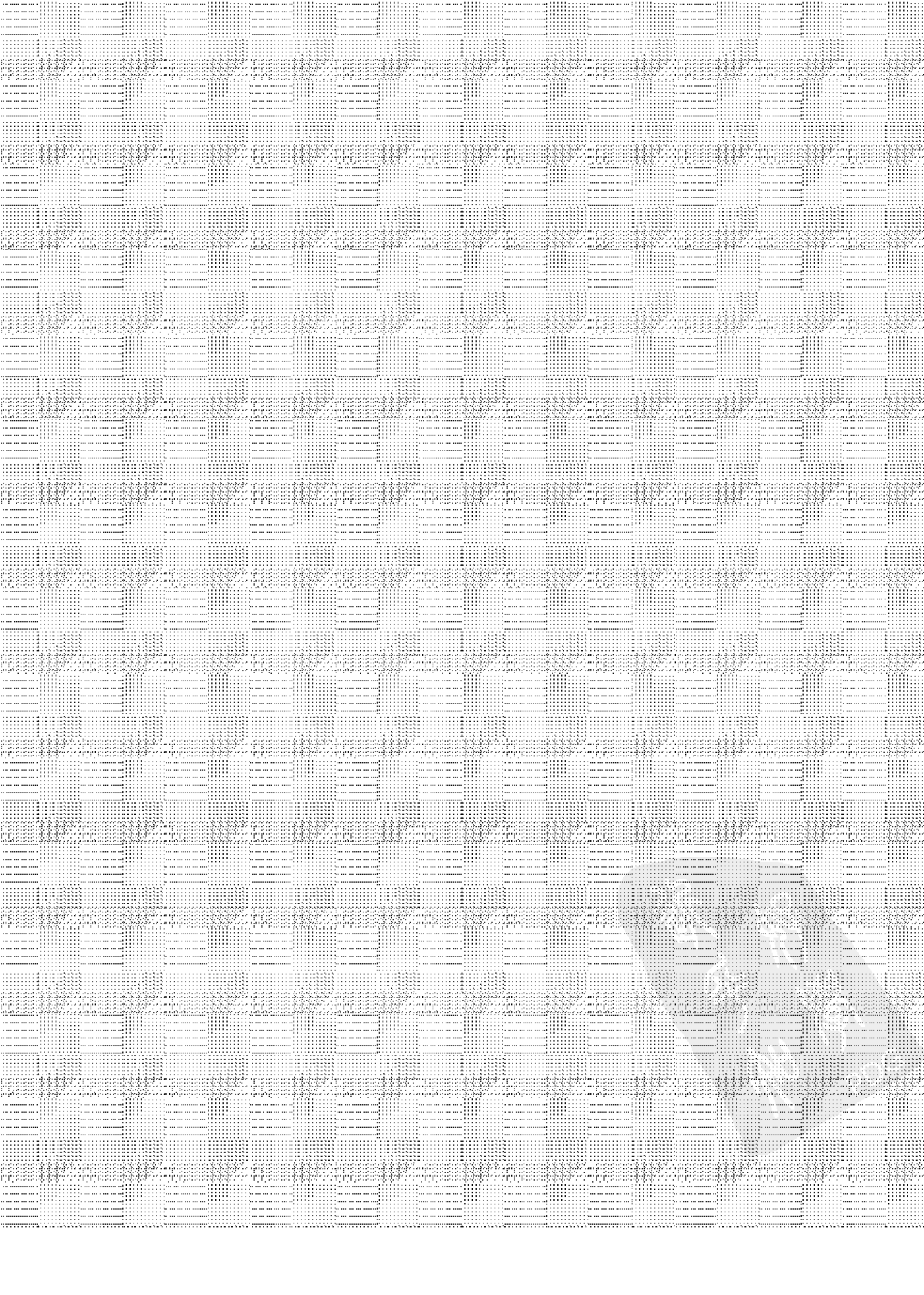
Joel 还没来得及说什么，他的老板就说话了："那个云方案的计划你做得很好，那正是我们所需要的东西。今天下午我将参加一个董事会议。" Summerson 先生做了一个投掷飞盘的动作，把 Joel 的提案扔到他的桌子上。

"我做了些笔注，希望你准备十几页的幻灯片，12:30 左右来会议室。"

Joel 惊恐地看着那些红色字迹。"额，先生？"小心翼翼地说。

"没什么大不了的，Joel。大多数董事会成员都曾经是像我们一样的普通人。我想你喜欢匹萨吧。"

Joel 思索了一下"我们"这个词。他正要问点什么，可是 Summerson 已经走了。Joel 坐下来，看了看身上的 polo 衬衫和牛仔裤。他在想回家换身衣服还能不能赶上这个报告。



第15章

MySQL集群

一阵平缓的敲门声提醒Joel有人来了，他一抬头看见满脸愁容的Summerson先生。

“这次要靠你了，我们遇到麻烦了。”

Joel什么也没说，他在想“靠你”是什么意思。目前Summerson先生已经给他安排了很多紧张的工作。

“我们刚刚得知，有一个新客户想在实时和‘五个九’的环境中使用我们最新的数据库应用程序。”

“始终运行、不停机？”

“是的。我知道MySQL很可靠，但我们现在没时间把应用程序的后台改为容错的数据库服务器。”

Joel记得书里面有一章介绍了MySQL，不知道能不能派得上用场。他决定试一试：“我们可以采用集群技术。”

“集群？”

“是的，MySQL有一个集群版本，它是一个容错的数据库系统。我记得它可以在某些相当苛刻的环境下工作，像远程通信等……”

Summerson先生眼前一亮，仿佛站得更直了，接着他做了一个决定性的发言。

“太好了，明天早上给我做个报告。我希望成本和硬件方面的需求和限制都能够满足，

不要有所保留。如果这么做行得通的话那就去做，但我不想凭感觉冒险。”

“我马上就去做。”Joel 不知道自己怎么就被卷了进来。Summerson 先生离开后，他叹了口气，然后打开心爱的 MySQL 书。

“这可能是我目前遇到的最大挑战了。”他说。

高性能、高可用性、冗余和可扩展性都是数据库规划时需要考虑的重要因素，常常采用商用高可用性硬件和均衡负载等方案来寻求改善复制拓扑的方法。尽管这样常常可以满
526 足大部分需求，但是如果需要无单点故障的方案，而且要求极高的吞吐量，上线时间达到 99.999%，那么可以考虑使用 MySQL 集群技术。

本章将介绍 MySQL 集群技术的相关概念，演示如何启动和停止简单集群，讨论 MySQL 集群使用过程中的关键问题，包括高可用性、分布式数据和数据复制。首先将解释什么是 MySQL 集群，以及它与普通的 MySQL 服务器有何不同。

什么是MySQL集群

MySQL 集群是一个无共享的（shared-nothing）、分布式节点架构的存储方案，其目的是提供容错性和高性能。数据在单个数据节点（有时也称存储节点）上存储和复制，每个数据节点运行在独立的服务器上并维护数据的一份拷贝。每个集群还有管理节点。数据更新使用读已提交隔离级别（read-committed isolation）来保证所有节点数据的一致性，使用两阶段提交机制（two-phased commit）保证所有节点都有相同的数据（如果任何一个写操作失败，则更新失败）。

MySQL 集群的最初实现将所有信息都保存在主存内，没有任何永久性存储。后来 MySQL 集群允许数据存储在磁盘上。通过存储引擎层将 MySQL 服务器作为查询引擎，可以使 MySQL 集群的性能达到最佳。这样就可以将 MySQL 应用透明地迁移到 MySQL 集群中去。

无共享的对等节点使得某台服务器上的更新操作在其他服务器上立即可见。传播更新使用一种复杂的通信机制，这一机制专用来提供跨网络的高吞吐量。该架构通过多个 MySQL 服务器分配负载，从而最大程度地达到高性能，通过在不同位置存储数据保证高可用性和冗余。

术语和组件

MySQL 集群的典型部署是在某个网络的不同机器上安装集群组件。因此，MySQL 集群

又称网络数据库（network database，NDB）。这里“MySQL 集群”指的是 MySQL 服务器和 NDB 组件，而“NDB”或“NDB 集群”则特指集群组件。

MySQL 集群是一个数据库系统，使用 MySQL 服务器作为前端来支持标准的 SQL 查询。名为 DNBcluster 的存储引擎是连接 MySQL 服务器和集群技术的接口，这个关系经常容易混淆。如果没有 NDB 集群组件，就不能使用 NDBcluster 存储引擎。

但是，没有 MySQL 服务器也可以使用 NDB 集群技术，只不过需要一些 NDB API 的底层编程。 527

NDB API 是面向对象的，实现了索引、扫描、事务和事件处理。你可以编写检索、存储和管理集群数据的应用。NDB API 还提供了面向对象的错误处理机制，允许有序的关机和故障恢复。如果你是一个开发者，想了解更多关于 NDB API 的信息，请参见 MySQL NDB API 在线文档（*http://dev.mysql.com/doc/NDBapi/en/index.html*）。

MySQL集群和MySQL有何不同

你可能会问：“集群和复制之间有什么区别呢？”集群的定义很多，通常认为集群包含成员、消息、冗余和自动故障转移等功能，而复制仅仅是一个服务器向另一个服务器发送消息（数据）的方式。我们先讨论集群内部的复制（又称本地复制），后面再详细讲述 MySQL 复制。

典型配置

MySQL 集群有如下三层。

- 应用程序层：负责与 MySQL 服务器通信的各种应用程序。
- MySQL 服务器层：处理 SQL 命令，并与 NDB 存储引擎通信的 MySQL 服务器。
- NDB 集群组件层：NDB 集群组件有时也称数据节点，负责处理查询，然后将结果返回给 MySQL 服务器。



每一层都可以独立地纵向扩展（scale up），即通过更多的服务器进程来提高性能。

图 15-1 显示了典型的集群配置概念图。

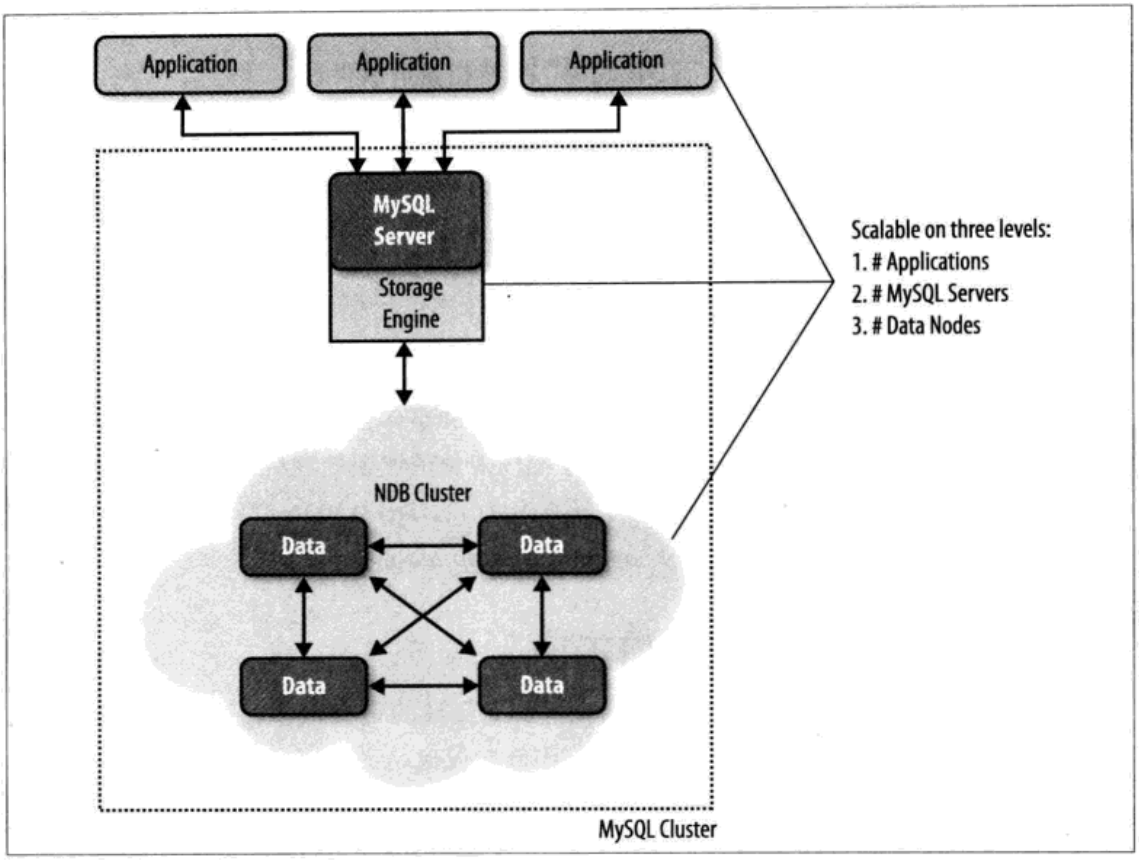


图15-1：MySQL集群

应用程序连接到MySQL服务器，通过存储引擎层（如NDB存储引擎）访问NDB集群组件。接下来将详细讨论NDB集群组件。

配置的种类很多。可以使用多个MySQL服务器来连接单个NDB集群，甚至可以通过MySQL复制连接到多个NDB集群。后面再讨论这些配置。

528

MySQL集群的特点

为了实现最高性能、高可用性和冗余等目标，数据在集群内部的对等数据节点之间相互复制。数据复制采用同步机制，每个数据节点连接到所有其他数据节点上，数据在多个数据节点上存储。



集群之间也可以复制数据，这时需要使用MySQL复制技术，它是异步的。前面的章节讲过，异步复制意味着更新Slave的时候有一定延迟，Slave不会将更新的进度报告给Master，所以你不能像在单个MySQL集群中那样期望复制架构中的所有服务器之间完全一致。

MySQL 集群有一些创建高可用性系统的专用功能，主要包括：

529

- 节点恢复
 通过通信丢失或心跳失效来检测数据节点故障，还可以将节点配置为从其他节点的数据拷贝中自动重启。故障和恢复可以包含单个或多个存储节点。又称本地恢复。
- 日志
 通常数据更新时，向每个数据节点的日志中写入一份数据变化事件的拷贝。该日志可以将数据恢复到某个时间点。
- 检查点
 集群支持两种检查点，即本地检查点和全局检查点。本地检查点删除日志文件的尾部。当所有数据节点上的日志都被刷新（flush）到磁盘时将创建全局检查点，在磁盘上创建一个所有节点数据的事务一致性快照。通过这种方式，检查点允许整个系统从某个已知的同步点恢复所有节点。
- 系统恢复
 如果整个系统非正常关闭，可以使用检查点和变化日志进行恢复。通常从某个已知的同步点将数据从磁盘复制到内存。
- 热备份及恢复
 可以在不干扰事务的情况下创建每个数据节点的同步备份，包括数据库中对象的元数据、数据本身和当前事务日志。
- 无单点故障
 此架构保证了任何节点的失效都不会导致数据库系统崩溃。
- 故障转移
 为了保证节点可恢复，所有事务的提交都采用读已提交隔离级别和两阶段提交机制。这样事务就是双重安全的，也就是说，事务到达用户之前存储在两个不同的地方。
- 分区
 数据在数据节点之间被自动分区。MySQL 5.1 版本集群支持用户自定义分区。
- 联机操作
 可以无中断地联机执行很多维护操作。通常需要正常关闭服务器或者给数据加锁。例如，联机添加新的数据节点，改变表结构，甚至重组集群中的数据。

关于 MySQL 集群的更多信息，参见在线参考手册（*http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/index.html*）。

局部和全局冗余

530

使用两阶段提交协议可以创建局部冗余（在某个集群内部）。原则上，如果每个节点都同意更改，则提交事务。在同意阶段，每个节点都保证下一轮有足够的资源来提交更改。

在NDB集群中，MySQL服务器的提交协议允许多个节点更改，NDB集群也有个两阶段提交的优化版本，减少了同步复制过程中发送的消息量。这种两阶段协议保证数据冗余地存储在多个数据节点上，这种状态称为局部冗余（*local redundancy*）。

全局冗余（*global redundancy*）在集群之间使用MySQL复制，这会在整个复制拓扑中建立两个节点。前面讲过MySQL复制是异步的，因为在复制事件到来或执行的时候没有确认。图15-2解释了局部冗余和全局冗余的区别。

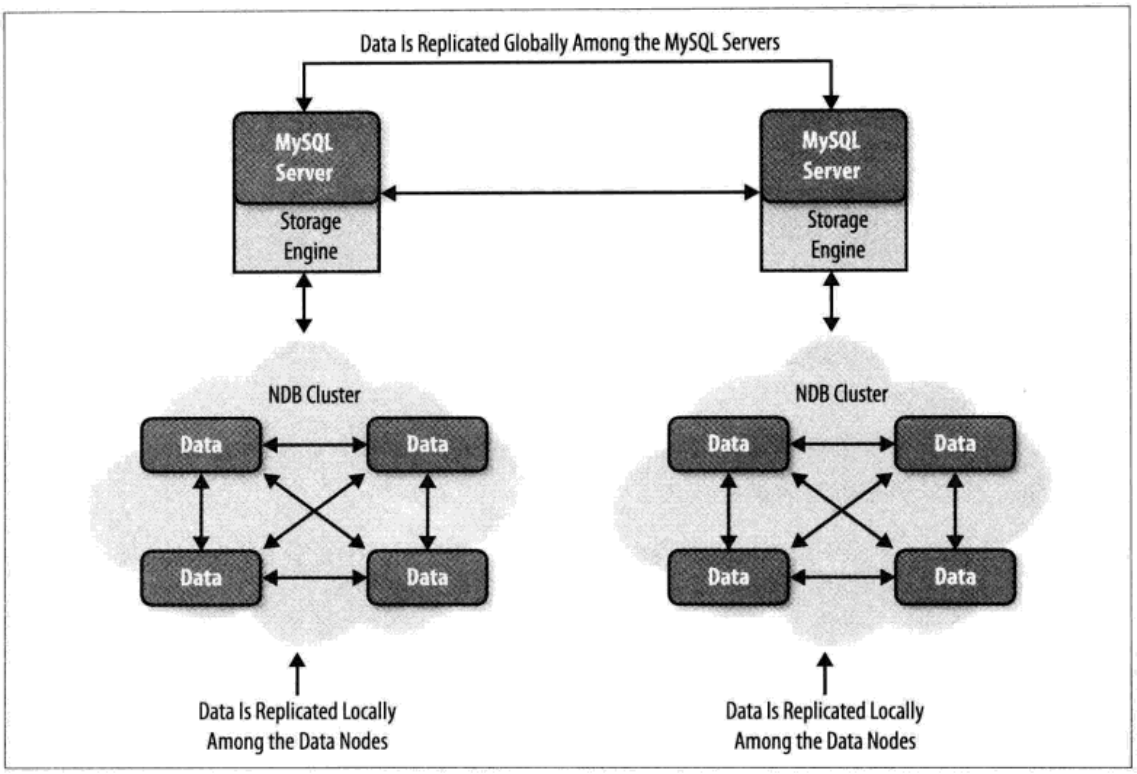


图15-2：局部冗余和全局冗余

531

日志处理

MySQL集群实现了两种类型的检查点：(1) 本地检查点，用于清除部分重做日志；(2) 全局检查点，主要用于不同数据节点之间的同步。全局检查点对于复制来说很重要，因为它形成了事务组之间的边界，称为*epoch*。每个epoch是集群之间复制的单位。实际上，MySQL复制把两个连续的全局检查点之间的事务组看成单个事务。

冗余和分布式数据

数据冗余用副本（*replica*）实现，每个副本包含数据的一份拷贝。这样集群就可以容错：

如果任何一个节点失效，仍然可以访问数据。当然，集群中的副本越多，其容错性就越好。

裂脑综合征

如果一个或多个节点失效，可能其他数据节点之间也不能通信，这样两组数据节点就处于“裂脑”状态。这类情况很麻烦，因为从理论上说，每组数据节点都成了一个独立集群。

为此，需要一个网络分区算法解决各组数据节点之间的竞争，每组独立进行选举。节点数目较少的组将重启，然后分别将该组中的每个节点添加到节点数目较多的组。

如果两组节点数目相当，仍然不能解决问题。例如将四个节点分成两组，每组两个节点，怎么选择组呢？这时可以定义一个仲裁器，规定第一个成功连接到仲裁器的组获胜。

仲裁器可以是MySQL服务器（SQL节点）或管理节点。为了达到高可用性，最好将仲裁器放在非数据节点的系统上。

带有仲裁器的网络分区算法在MySQL集群中是完全自动化的，“少数”的定义与节点组相关，与仅对节点进行计数的方法相比，它的可用性更高。

可以指定集群中的数据副本数目（`NoOfReplicas`）。需要多少replica，就设置相应数量的数据节点。还可以利用分区将数据分布到各个数据节点，每个数据节点仅存储部分数据，这样查询更快。由于数据有多份拷贝，即使节点故障仍可以查询数据，也可以恢复丢失的节点（因为其他replica中数据还存在）。为此，每一个replica都需要多个数据节点来存储。例如，如果有两个replica，且已分区，那么你至少需要四个数据节点（每个replica需要两个数据节点）。 532

MySQL集群的架构

MySQL集群由一个或多个MySQL服务器组成，MySQL服务器通过NDB存储引擎与NDB集群通信。NDB集群由以下组件构成：存储和检索数据的数据或存储节点，以及一个或多个管理节点，负责协调数据节点的启动、关闭和恢复。大部分NDB组件都作为守护进程执行，MySQL集群还提供操作守护进程的客户端实用程序。下面列出了一些守护进程和实用程序。图15-3描绘了这些组件相互之间是如何通信的。

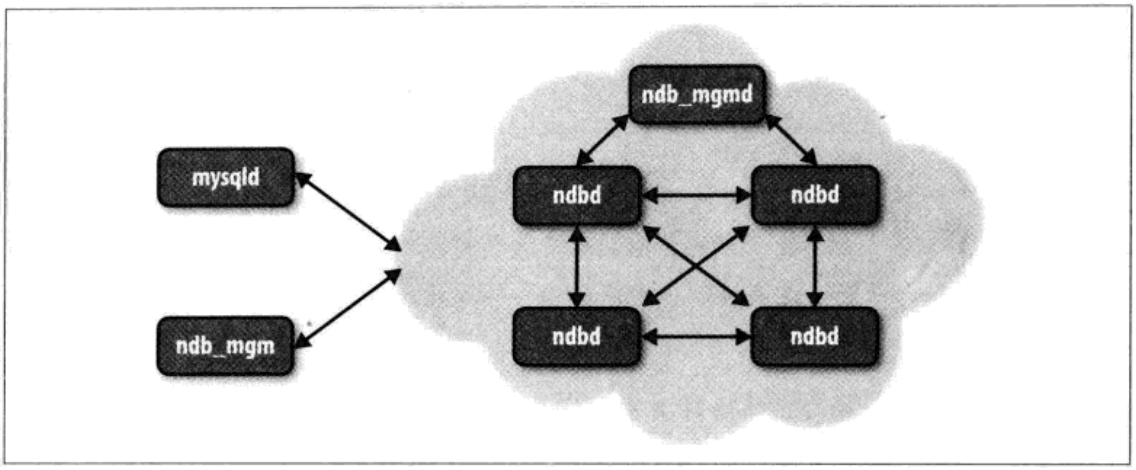


图15-3：My SQL集群组件

- *mysqld*
 MySQL 服务器。
- *NDBd*
 数据节点。
- *NDBmtd*
 多线程数据节点。
- *NDB_mgmd*
 集群的管理服务器。
- *NDB_mgm*
 集群的管理客户端。

每个名为 *mysqld* 的 MySQL 服务器通常都支持一个或多个 SQL 查询应用，然后从数据节点返回数据。讨论 MySQL 集群的时候，有时 MySQL 服务器又称 SQL 节点。

数据节点是一系列 NDB 守护进程，负责存储和检索内存或磁盘上（由配置决定）的数据。数据节点安装在集群中的各个服务器上。还有一个名为 NDBmtd 的多线程数据节点守护进程，运行在支持多 CPU 核的平台上。如果在现代的多核 CPU 专用服务器上使用多线程数据节点，可以提高数据节点的性能。

管理守护进程 NDB_mgmd 运行在服务器上，负责读入配置文件，然后将信息分发到集群中的所有节点上。NDB 管理客户端实用程序 NDB_mgm 可以检查集群的状态，开始备份，然后执行其他管理功能。这个客户端运行在一个方便管理的主机上，并与守护进程通信。

很多实用程序可以简化维护工作，下面给出了几个常用的实用程序。要想得到完整列表，

请查阅 NDB 集群文档。

- *NDB_config* <533>
 抽取已有节点的配置信息。
- *NDB_delete_all*
 删除 NDB 表中的所有行。
- *NDB_desc*
 描述 NDB 表（就像 SHOW CREATE TABLE）。
- *NDB_drop_index*
 删除 NDB 表的索引。
- *NDB_drop_table*
 删除 NDB 表。
- *NDB_error_reporter*
 诊断集群中的错误和问题。
- *NDB_redo_log_reader*
 检查并输出集群的重做日志。
- *NDB_restore*
 执行集群恢复，使用 NDB 管理客户端进行备份。

如何存储数据

MySQL 集群将所有的索引列都保存在主存中，其他非索引列可以存储在内存中，或者存储到带有内存页面缓存的磁盘上。磁盘可以比内存存储更多的非索引列数据。

如果数据发生改变（通过 INSERT, UPDATE, DELETE 等），MySQL 集群将发生改变的记录写入重做日志，然后通过检查点定期将数据写入磁盘。前面讲过，日志和检查点可以从 <534> 磁盘进行故障恢复。但是，由于重做日志是异步提交的，所以故障期间可能有少量事务丢失。为了减少事务丢失，MySQL 集群实现了延迟写入（默认延迟两秒，可配置），这样就可以在故障发生时完成检查点写入，而不会丢失最后一个检查点。一般单个数据节点故障不会导致任何数据丢失，因为集群内部采用同步数据复制。

MySQL 集群的表在内存中维护，只有在向重做日志中写入数据并执行必要的检查点操作时，集群才需要访问磁盘存储。因为写日志和检查点都是顺序操作，很少出现随机访问模式，所以比起关系数据库中传统的磁盘缓存机制，MySQL 集群可以通过有限的磁盘硬件得到更高的写吞吐率。

使用下面的公式可以计算出一个数据节点需要的内存大小。数据库的大小是行的大小乘以每个表的行数。记住如果使用磁盘存储非索引字段，在计算内存需求时只需计算索引

字段。

（数据库的大小 *replica 的数量 *1.1）/ 数据节点的数量

这是用于粗略计算的一个简单公式。在规划集群的内存需求时，可以考虑查阅在线 MySQL 集群参考手册。

还可以在大部分分发工作中使用 Perl 脚本 *NDB_size.pl*，连接到一个正在运行的 MySQL 服务器，遍历数据库中已有的表，然后计算 MySQL 集群可能需要的内存。这个脚本很方便，因为它可以让你先在一个普通 MySQL 服务器上创建和填充表，然后检查内存配置，配置系统，最后将数据装载到集群中去。这同样有利于定期运行，从而避免数据库模式改变带来的内存问题，且让你了解内存的使用情况。例 15-1 描述了含有一张表的简单数据库的报表示例。为了得出数据库的总大小，我们用汇总数据中的数据行大小乘以行数。数据和索引每行大小为 84 字节（MySQL 版本为 5.1），若有 64 000 行，则需要 5 376 000 字节的内存来存储这张表。



如果脚本产生了关于丢失 *Class/MethodMaker.pm* 模块的错误，则需要在自己的系统上安装这个类。例如，在 Ubuntu 上使用如下命令安装：

```
sudo apt-get install libclass-methodmaker-perl
```

535 例15-1：使用NDB_size.pl检查数据库的大小

```
cbell@cbell-mini:~/mysql-cluster-gpl-7.0.13-linux-i686-glibc23/bin$ ./NDB_size.pl \
--database=cluster_test --user=root
NDB_size.pl report for database: 'cluster_test' (1 tables)
----------------------------------------------------------
Connected to: DBI:mysql:host=localhost

Including information for versions: 4.1, 5.0, 5.1

cluster_test.City
-----------------

DataMemory for Columns (* means varsized DataMemory):
      Column Name  Type  Varsized   Key    4.1   5.0   5.1
         district        char(20)           20    20    20
       population         int(11)            4     4     4
            ccode         char(3)            4     4     4
             name        char(35)           36    36    36
               id         int(11)   PRI      4     4     4
                                           --    --    --
Fixed Size Columns DM/Row                  68    68    68
 Varsize Columns DM/Row                     0     0     0
```

```
DataMemory for Indexes:
       Index Name       Type         4.1      5.0      5.1
          PRIMARY       BTREE        N/A      N/A      N/A
                                      --       --       --
       Total Index DM/Row              0        0        0

IndexMemory for Indexes:
                Index Name         4.1        5.0        5.1
                   PRIMARY          29         16         16
                                    --         --         --
             Indexes IM/Row         29         16         16

Summary (for THIS table):
                                   4.1        5.0        5.1
    Fixed Overhead DM/Row           12         12         16
           NULL Bytes/Row            0          0          0
           DataMemory/Row           80         80         84
(Includes overhead, bitmap and indexes)

Varsize Overhead DM/Row             0          0          8
   Varsize NULL Bytes/Row           0          0          0
       Avg Varside DM/Row           0          0          0

                 No. Rows           3          3          3

        Rows/32kb DM Page         408        408        388
Fixedsize DataMemory (KB)          32         32         32

Rows/32kb Varsize DM Page           0          0          0
  Varsize DataMemory (KB)           0          0          0

         Rows/8kb IM Page         282        512        512

         IndexMemory (KB)           8          8          8

Parameter Minimum Requirements
------------------------------
* indicates greater than default

            Parameter     Default       4.1       5.0       5.1
      DataMemory (KB)       81920        32        32        32
   NoOfOrderedIndexes         128         1         1         1
          NoOfTables          128         1         1         1
     IndexMemory (KB)       18432         8         8         8
NoOfUniqueHashIndexes          64         0         0         0
       NoOfAttributes        1000         5         5         5
         NoOfTriggers         768         5         5         5
```

536

例 15-1 以一个很简单的表为例，不仅输出了行的大小，还有数据库中表的统计数据。这

个报告还给出了索引统计数据，索引是集群高性能的关键机制。

这个脚本也展示了 MySQL 不同版本之间不同的内存需求。如果你用的是老版本的 MySQL 集群，就会看到其中的区别。

分区

数据分区是 MySQL 集群的一个重要方面。MySQL 集群将数据水平分区，即使用某种函数将行自动分布到各个数据节点上，基于表关键字上的哈希算法。在早期的 MySQL 版本中，软件使用一种内部分区机制，而 MySQL 5.1 及其后的版本允许提供自定义函数进行数据分区。如果你使用自己的分区函数，就要创建一个函数来保证数据平均分布到各个数据节点之间。



如果表没有主键，MySQL 集群就会添加一个代理主键。

分区可以使 MySQL 集群获得更好的查询性能，因为分区支持数据节点之间的分布式查询。因此，在几个节点之间收集数据时，查询的返回结果会比单个节点更快。例如，在每个数据节点上执行下面的查询，对每个节点上的列求和，然后汇总：

```
SELECT SUM(population) FROM cluster_db.city;
```

537 数据分布在数据节点之间，如果数据有多个 replica（副本），就可以防止故障的发生。如果你想使用分区将数据分布到多个数据节点上，还要保证每行至少有两份 replica，这样集群才是容错的。

事务管理

MySQL 集群不同于 MySQL 服务器的另一个方面就是关于事务型数据操作。前面提过，MySQL 集群协调数据节点之间的事务性变化。包括两个子过程，即事务协调器（*transaction coordinator*）和本地查询处理器（*local query handler*）。

事务协调器处理全局的分布式事务和其他数据操作。本地查询处理器管理集群数据节点本地的数据和事务，并协调处理数据节点的两阶段提交。

每个数据节点都可以是事务协调器（可以对其进行调整）。当应用程序执行一个事务时，集群就会连接到其中一个数据节点的事务协调器，默认选择最近的数据节点。如果有多个同样距离的可用连接，就采用轮转法（round-robin）选择事务协调器。

接下来，被选中的事务协调器向每个数据节点发送查询，然后各个节点的本地查询处理器执行这个查询，并与事务协调器一起协调两阶段提交。一旦所有数据节点都验证了这个事务，事务协调器就会提交这个事务。

MySQL 集群支持 read-committed 事务隔离级别，即如果事务执行期间发生改变，只有已提交的改变在事务过程中是可读的。这样，MySQL 集群保证了事务过程中的数据一致性。

要想了解更多关于事务在 MySQL 集群是如何工作的，以及事务的重要局限性，参见在线 MySQL 参考手册中的 MySQL 集群这一章。

联机操作

在 MySQL 5.1 及以后的版本中，可以联机执行某些操作，也就是说不需要关闭服务器，也不用将系统或数据库部分加锁。下面的列表简单讨论了 MySQL 集群可用的一些联机操作，并给出了相应的版本信息。

- 备份（5.0 及之后的版本）。
 可以使用 NDB 管理控制台来执行快照备份（非阻塞操作），从而创建集群中数据的一个备份。这个操作包括复制元数据（所有表的名字和定义）、表数据和事务日志（变化的历史记录）。这与 `mysql dump` 备份不同，因为它是非阻塞的，不使用表扫描来读取记录。可以使用特殊的 `NDB_restore` 实用程序来恢复数据。 538
- 添加和删除索引（5.1 及之后的版本）
 使用 `ONLINE` 关键字联机执行 `CREATE INDEX` 或 `DROP INDEX` 命令行。当请求联机操作时，这个操作是非复制的，即不会产生数据的拷贝来建立索引，这样以后就不需要重建索引。这样做的好处是在执行 alter table 操作的时候，正在被修改的表不会被加锁导致其他 SQL 节点不可读，从而事务可以继续进行。但是，对于执行 alter 操作的 SQL 节点上的其他查询来说，表仍然是锁定的。



在 MySQL 5.1.7 及以后的版本中，只有当索引列的宽度可变时，添加和删除索引操作才是联机执行的。

- 修改表（6.2 及以后的版本）
 使用 `ONLINE` 关键字联机执行 `ALTER TABLE` 语句。这同样也是非复制的，和联机添加索引的好处一样。此外，在 MySQL 7.0 及以后的版本中，可以使用 `REORGANIZE PARTITION` 命令跨分区联机重组数据，但是不能加 `INTO (partition_definitions)` 选项。



目前还不支持改变默认字段值或数据类型的联机操作。

- 添加数据节点和数据组（7.0 及以后的版本）
 可以联机管理数据节点扩张、横向扩展或故障之后的节点替换。这个过程在参考手册中有更加详细的描述。简单地说，包括修改配置文件，执行 NDB 管理守护进程的轮流重启，执行已有数据节点的轮流重启，启动新的数据节点，然后进行分区重组。

关于 MySQL 集群的更多信息及其架构和 7.0 版本的新特性，请见白皮书 *http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster7_architecture.php*.

539

配置实例

本节中将展示一个 MySQL 集群的配置实例，包括分别运行在两个系统上的两个数据节点，以及运行在第三个系统上的 MySQL 服务器和 NDB 管理节点。这个例子简化了数据节点的设置，如图 15-4 所示。

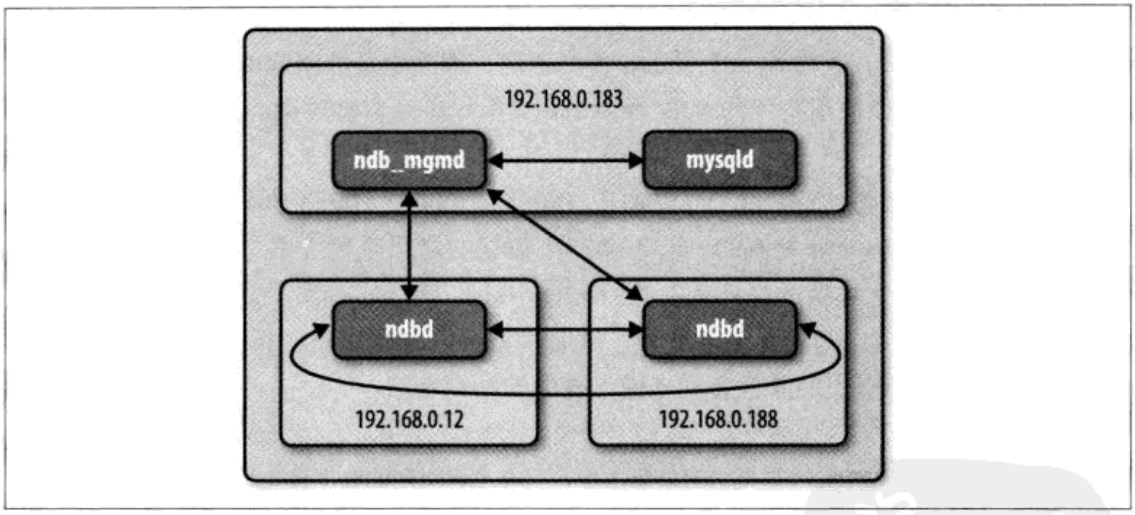


图15-4：集群配置实例

如图 15-4 所示，一个节点上既包含 NDB 管理守护进程，同时还是 SQL 节点（即 MySQL 服务器），还有两个数据节点，各自运行在自己的系统中。你需要至少三台计算机来完成这样一个基础的 MySQL 集群配置，以提高可用性或性能。

这是 MySQL 集群的最小配置。如果 replica 的数量设为 2，则可以为最小配置容错。如

果 replica 的数量设为 1，为获得更好的性能，配置可以支持分区，但却不能容错。

一般来说，同一个节点上同时运行 NDB 管理守护进程和 MySQL 服务器是允许的，但是如果数据节点的数目很多，或者你想保证最大容错，可能需要将这个守护进程迁移到另一个系统上去。

入门

从 MySQL 下载页面上可以获得 MySQL 集群，跟 MySQL 服务器一样，它也是开源的。可以下载二进制版本或者常见平台的安装文件，还可以下载源码，然后在自己的平台上构建集群。一定要检查主机操作系统需要注意的具体问题。 540

按照在线 MySQL 参考手册上的正常安装过程进行，除了特定目录以外，NDB 工具与 MySQL 服务器的二进制文件安装在同一位置。

在介绍例子之前，让我们先重温一下关于配置 MySQL 集群的常用概念。集群配置由 NDB 管理守护进程维护，从配置文件中读取（初始化）配置。可以使用多个参数调整集群的各个部分，但现在我们会集中考虑最小配置。

配置文件由多个部分组成，至少包含下面几项：

- *mysqld*
 MySQL 服务器和 SQL 节点的配置文件的常见配置项。
- *NDB DEFAULT*
 全局设置的默认配置项，用于指定应用于每个节点的所有设置，包括数据和管理。注意此项的名字中包含一个空格而不是下画线。
- *NDB_MGMD*
 用于 NDB 管理守护进程。
- *NDBD*
 必须为每个数据节点都加上该项。

例 15-2 显示了符合图 15-4 所示的配置的最小配置文件。

例15-2：最小配置文件

```
[NDBD DEFAULT]
NoOfReplicas= 2
DataDir= /var/lib/mysql-cluster

[NDB_MGMD]
Hostname=192.168.0.183
DataDir= /var/lib/mysql-cluster
```

```
[NDBD]
Hostname=192.168.0.12

[NDBD]
Hostname=192.168.0.188

[MYSQLD]
Hostname=192.168.0.183
```

541 这个例子给出了一个简单的双节点集群复制所需的最少变量。因此，NoOfReplicas 选项设置为 2。注意我们将 datadir 设为 /var/lib/mysql-cluster，也可以随便设置，但大多数 MySQL 集群都采用这个目录。

最后，注意我们给每个节点都指定了一个主机名，这很重要，因为 NDB 管理守护进程需要知道集群中所有节点的位置。如果你已经下载并安装了 MySQL 集群，下面就是对主机名做必要的更改，以配合我们的例子。

将集群配置文件放在 */var/lib/mysql-cluster* 目录下，并命名为 *config.ini*（这是集群配置文件的标准位置和名称）。



数据节点上并不需要完全安装 MySQL 集群二进制包，后面你就会发现，只需要 NDBd 守护进程即可。

启动MySQL集群

启动 MySQL 集群需要有序地执行一组命令。我们就用上面的例子逐步跟踪启动集群过程，不过首先简单看看这个过程的一般步骤：

1. 启动管理节点；
2. 启动数据节点；
3. 启动 MySQL 服务器（SQL 节点）。

本例中，我们先在 192.168.0.183 上启动 NDB 管理节点，然后（分别在 192.168.0.12 和 192.168.0.188 上）启动每个数据节点。数据节点开始运行后，在 192.168.0.183 上启动 MySQL 服务器。经过一段简短的启动延迟之后，集群就可以使用了。

启动管理节点

第一个要启动的节点是名为 NDB_mgmd 的 NDB 管理守护进程，位于 MySQL 安装目录

的 *libexec* 文件夹下。例如，它在 Ubuntu 上的位置为 */usr/local/mysql/libexec*。

通过超级用户启动 NDB 管理守护进程，并指定 `--initial` 和 `-f` 选项。`--initial` 选项告诉集群这是第一次启动，需要清除以前启动时存储的配置信息。`-f` 选项告诉守护进程配置文件的位置。例 15-3 显示了如何启动 NDB 管理守护进程。

例15-3：*启动NDB管理守护进程*

```
cbell@mysql-xps-400:/usr/local/mysql/bin$ sudo ../libexec/NDB_mgmd --initial \
-f /var/lib/mysql-cluster/config.ini
2010-03-25 09:10:28 [MgmtSrvr] INFO
     -- NDB Cluster Management Server. mysql-5.1.44 NDB-7.0.14
2010-03-25 09:10:29 [MgmtSrvr] INFO
     -- Reading cluster configuration from '/var/lib/mysql-cluster/config.ini'
```

542

启动时最好提供 `-f` 选项，因为有些安装中的配置文件搜索模式具有不同的默认位置。使用命令 `NDB_mgmd--help`，并查找“Default options are read from”，你就可以查看其默认安装位置。而以后的守护进程启动就不需要再指定 `-f` 选项了。

启动管理控制台

现在启动 NDB 管理控制台，并检查 NDB 管理守护进程是否正确地读取配置。虽然这一步不是必需的，但最好执行。NDB 管理控制台的名字为 NDB_mgm，位于 MySQL 安装目录的 *bin* 目录下。使用 `SHOW` 命令可以查看配置，如例 15-4 所示。

例15-4：*初始启动NDB管理控制台*

```
cbell@mysql-xps-400:/usr/local/mysql/bin$ ./NDB_mgm
-- NDB Cluster -- Management Client --
NDB_mgm> SHOW
Connected to Management Server at: 192.168.0.183:1186
Cluster Configuration
---------------------
[NDBd(NDB)]     2 node(s)
id=2 (not connected, accepting connect from 192.168.0.188)
id=3 (not connected, accepting connect from 192.168.0.12)

[NDB_mgmd(MGM)]     1 node(s)
id=1    @192.168.0.183  (mysql-5.1.44 NDB-7.0.14)

[mysqld(API)]     1 node(s)
id=4 (not connected, accepting connect from 192.168.0.183)

NDB_mgm>
```

这个命令显示了数据节点及其 IP 地址，还有 NDB 管理守护进程和 SQL 节点。这时检查所有节点的 IP 地址是否配置正确，以及所有数据节点是否正确装载。如果更改了集群配

置却在这里看到旧值，可能是 NDB 管理控制台还没有读取到新的配置文件。

这个输出告诉我们 NDB 管理控制台已经装载成功并准备就绪。如果还没有，SHOW 命令会出现通信错误而失败。如果出现这个错误，首先检查 NDB 管理客户端与 NDB 管理守
543 护进程是否运行在同一个服务器上。如果不是，使用 --NDB-connectstring 选项，并提供 IP 地址或 NDB 管理守护进程所在机器的主机名作为参数。

最后，注意节点的节点 ID。从 NDB 管理控制台发出命令指定集群中的节点时，需要这个信息。任何时候使用 HELP 命令都可以查看其他可用的命令信息。还需要知道 SQL 节点的节点 ID，这样可以正确地启动它们。



集群中的每个节点都要在 *config.ini* 文件中使用 --NDB-nodeid 参数指定其节点 ID。

还可以用 STATUS 命令查看节点状态，ALL STATUS 命令可以查看所有节点的状态，*node-id* STATUS 查看特定节点的状态。这个命令用于监视集群的启动，因为输出报告了数据节点正处于哪个启动阶段。关于数据节点启动阶段的更多细节请参考在线 MySQL 参考手册的 MySQL 集群部分。

启动数据节点

既然我们已经启动了 NDB 管理守护进程，现在该启动数据节点了。但是，在这之前，让我们先看看创建一个 NDB 数据节点的最小需求。

为了创建 NDB 数据节点，你需要为目标宿主操作系统编制一个 NDB 数据节点守护进程（即 NDBd）。首先，创建文件夹 */var/lib/mysql-cluster*，然后将 NDBd 可执行文件复制进来，完成！显然，这样很容易为数据节点的创建编写脚本。

可以使用 --initial-start 选项启动数据节点（NDBd），表明这是第一次启动集群。还必须提供 --NDB-connectstring 选项，加上 NDB 管理守护进程的 IP 地址作为参数。例 15-5 给出了第一次启动数据节点的示例，每个节点都执行这样的操作。

例15-5：*启动数据节点*

```
cbell@mysql-mini:~/mysql-cluster-gpl-7.0.13-linux-x86_64-glibc23/bin$
 sudo ./NDBd --initial-start --NDB-connectstring=192.168.0.183
2010-03-25 09:04:18 [NDBd] INFO
     -- Configuration fetched from '192.168.0.183:1186', generation: 1
```

如果你正在启动新的数据节点，需要重置数据节点，或者进行故障恢复，你可以指定 --initial 选项来强制数据节点清除已有的配置信息和缓存数据，然后从 NDB 管理守

护进程请求一份新的拷贝。



小心使用 --initial 选项，它们真的会删除你的数据！ <544>

返回管理控制台，然后检查状态（如例 15-6 所示）。

例15-6：数据节点的状态

```
NDB_mgm> SHOW
Cluster Configuration
---------------------
[NDBd(NDB)]    2 node(s)
id=2    @192.168.0.188  (mysql-5.1.41 NDB-7.0.13, Nodegroup: 0, Master)
id=3    @192.168.0.12  (mysql-5.1.41 NDB-7.0.13, Nodegroup: 0)

[NDB_mgmd(MGM)]    1 node(s)
id=1    @192.168.0.183  (mysql-5.1.44 NDB-7.0.14)

[mysqld(API)]    1 node(s)
id=4 (not connected, accepting connect from 192.168.0.183)
```

可以看到数据节点已经成功启动，因为已经显示了守护进程信息。还可以发现其中一个节点被选为集群复制的 Master。因为配置文件中 replica 的数目设为 2，所以我们将会有两份数据的拷贝。不要将这里的 Master 概念与 MySQL 复制中的 Master 混淆，后面将详细讨论两者的差别。

启动SQL节点

一旦数据节点开始运行，就可以连接 SQL 节点。要使 MySQL 服务器连接到 NDB 集群，必须指定几个选项。大部分人在 *my.cnf* 中指定这些选项，当然也可以在启动服务器时用命令行指定。

- `NDBcluster`
 告诉服务器你想加一个 NDB 集群存储引擎。
- `NDB_connectstring`
 告诉服务器 NDB 管理守护进程的位置。
- `NDB_nodeid and server_id`
 一般设为节点 ID，可以在管理控制台中用 `SHOW` 命令输出节点 ID 信息。

例 15-7 给出了集群中 SQL 节点的正确启动顺序。

545 例15-7：启动SQL节点

```
cbell@mysql-xps-400:/usr/local/mysql/bin$ sudo ../libexec/mysqld -NDBcluster \
--console -umysql
100325  9:14:21 [Note] Plugin 'FEDERATED' is disabled.
100325  9:14:21  InnoDB: Started; log sequence number 0 1112278176
100325  9:14:21 [Note] NDB: NodeID is 4, management server '192.168.0.183:1186'
100325  9:14:22 [Note] NDB[0]: NodeID: 4, all storage nodes connected
100325  9:14:22 [Note] Starting Cluster Binlog Thread
100325  9:14:22 [Note] Event Scheduler: Loaded 0 events
100325  9:14:23 [Note] NDB: Creating mysql.NDB_schema
100325  9:14:23 [Note] NDB: Flushing mysql.NDB_schema
100325  9:14:23 [Note] NDB Binlog: CREATE TABLE Event: REPL$mysql/NDB_schema
100325  9:14:23 [Note] NDB Binlog: logging ./mysql/NDB_schema (UPDATED,USE_WRITE)
100325  9:14:23 [Note] NDB: Creating mysql.NDB_apply_status
100325  9:14:23 [Note] NDB: Flushing mysql.NDB_apply_status
100325  9:14:23 [Note] NDB Binlog: CREATE TABLE Event: REPL$mysql/NDB_apply_
status
100325  9:14:23 [Note] NDB Binlog: logging ./mysql/NDB_apply_status
(UPDATED,USE_WRITE)
2010-03-25 09:14:23 [NdbApi] INFO     -- Flushing incomplete GCI:s < 65/17
2010-03-25 09:14:23 [NdbApi] INFO     -- Flushing incomplete GCI:s < 65/17
100325  9:14:23 [Note] NDB Binlog: starting log at epoch 65/17
100325  9:14:23 [Note] NDB Binlog: NDB tables writable
100325  9:14:23 [Note] ../libexec/mysqld: ready for connections.
Version: '5.1.44-NDB-7.0.14-debug'  socket: '/var/lib/mysql/mysqld.sock'
port: 3306  Source distribution
```

输出中包括了关于NDB集群连接、日志和状态等额外的注释信息。如果没有看到注释，或是发现了错误，首先检查SQL节点的启动选项是否正确的。指定节点ID和管理服务器的消息特别重要。如果有多个管理服务器正在运行，那么要保证SQL节点与正确的服务器通信。

SQL节点正确启动之后，返回管理控制台，检查所有节点的状态（如例15-8所示）。

例15-8：正在运行的集群的状态示例

```
NDB_mgm> SHOW
Cluster Configuration
---------------------
[NDBd(NDB)]    2 node(s)
id=2    @192.168.0.188  (mysql-5.1.41 NDB-7.0.13, Nodegroup: 0, Master)
id=3    @192.168.0.12  (mysql-5.1.41 NDB-7.0.13, Nodegroup: 0)

[NDB_mgmd(MGM)]    1 node(s)
id=1    @192.168.0.183  (mysql-5.1.44 NDB-7.0.14)

[mysqld(API)]    1 node(s)
id=4    @192.168.0.183  (mysql-5.1.44 NDB-7.0.14)
```

所有节点都已经连接上且正常运行。如果看到了更多这里没有显示的信息，那么你在启动节点的过程中就遇到了问题。检查每个节点的日志信息，确定哪里出错。最常见的问题是网络连接故障（如防火墙等）。NDB 节点默认使用端口 1186。 546

数据节点和 NDB 管理守护进程的的日志文件位于数据目录下，SQL 节点的日志则位于 MySQL 服务器的常规位置。

集群测试

我们的例子中的集群已经在运行了，下面进行一个简单的测试（如例 15-9 所示），以确保可以使用 NDBcluster 存储引擎创建数据库和表。

例15-9：集群测试

```
mysql> create database cluster_db;
Query OK, 1 row affected (0.06 sec)

mysql> create table cluster_db.t1 (a int) engine=NDBCLUSTER;
Query OK, 0 rows affected (0.31 sec)

mysql> show create table cluster_db.t1 \G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE 't1' (
  'a' int(11) DEFAULT NULL
) ENGINE=NDBcluster DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql> insert into cluster_db.t1 VALUES (1), (100), (1000);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from cluster_db.t1 \G
*************************** 1. row ***************************
a: 1
*************************** 2. row ***************************
a: 1000
*************************** 3. row ***************************
a: 100
3 rows in set (0.00 sec)
```

集群已经在运行中，可以装载数据和运行示例查询来进行试验。在数据更新过程中，请你让其中一个数据节点失效，然后重启它，看看单个数据节点的故障会不会影响整个系统的可访问性。

关闭集群

正如启动是有特定顺序的，关闭集群同样也是有序的：

547 1. 如果集群间有复制正在运行，先使 Slave 跟上进度，然后再停止复制；

2. 关闭 SQL 节点（mysql）；

3. 在 NDB 管理控制台中发出 SHUTDOWN 命令；

4. 退出 NDB 管理控制台。

如果 MySQL 复制运行在两个或更多的集群之间，首先要保证在关闭 SQL 节点之前，复制 Slave 赶上了 Master 的进度（即同步）。

在 NDB 管理控制台中发出 SHUTDOWM 命令，将会关闭所有的数据节点和 NDB 管理守护进程。

达到高可用性

使用高可用性的主要动机是保持服务可用。对于数据库系统来说，这就意味着我们必须保证数据总是可访问的。MySQL 集群就是用来满足这个需要的。MySQL 集群通过以下方式支持高可用性：数据节点之间的数据分布（减少单个节点的数据丢失风险），集群中的 replica 复制，丢失的数据节点的自动恢复（故障转移），通过心跳进行数据故障检测，以及使用本地和全局检查点来保证数据一致性等。

让我们先看看高可用性数据库系统的一些特点。高可用性的数据库系统必须满足以下要求：

- 99.999% 的上线率；
- 无单点故障；
- 故障转移；
- 容错。

99.999% 的上线时间意味着从实践的角度来说，数据总是可用的，即数据库服务器提供不间断的、连续的服务。假设前提是数据库永远不会由于组件故障或维护问题而离线。所有的操作，如维护和恢复，都是联机进行的，访问不会被中断。

这种理想情况的需求很少，只有最关键的行业才真正要求这样的质量。此外，还需要短期的例行程序及预防性的维护工作（从而达到将近 100% 的上线率）。有趣的是，关于可接受的上线时间的粒度是以上线率中 9 的个数来衡量的。表 15-1 给出了可接受的宕机时

间（离线时间）划分等级（单位：年）。

表 15-1：可接受的宕机时间表 548

Uptime	Acceptable downtime
99.000%	3.65 days
99.900%	8.76 hours
99.990%	52.56 minutes
99.999%	5.26 minutes

注意从这个表中可以看出，级别中的 9 数目越多，可接受的宕机时间就越少。如 99.999% 的上线率，意味着该系统在 1 年时间内，除了极少时间外，都要无间断地联机执行所有的维护操作。MySQL 集群可以通过很多种方式满足这种需求，包括数据节点轮流重启的能力、数据库的联机维护操作，以及多种数据访问渠道（通过 NDB API 连接的 SQL 节点和应用）等。

无单点故障意味着系统中不存在决定整个服务可用性的单个组件。通过为集群中每种类型的节点设置冗余，可以做到这一点。前面的例子中，我们有两个数据节点，因此可以防止单个数据节点故障。但是，我们只有一个管理节点和一个 SQL 节点，理想情况下，还需要添加额外的节点。MySQL 集群支持多个 SQL 节点，这样即使管理节点失效，集群仍然可以运行。

故障转移是指，如果一个组件失效，另一个组件可以代替它完成同样的功能。对 MySQL 数据节点来说，如果集群包含多个数据 replica，故障转移就可以自动执行。如果某个 replica 的一个 MySQL 数据节点失效，数据访问不会被中断。重启丢失的数据节点，它将从另一个 replica 中把数据拷贝回来。对 SQL 节点来说，由于数据实际是存储在数据节点上的，任何 SQL 节点之间都可以相互代替。

如果 NDB 管理节点发生故障，集群仍然可以继续运行，任何时候你都可以重新启动一个新的管理节点（在配置没有发生改变的情况下）。

可以使用前面章节讨论的常规高可用性解决方案，包括整个集群之间的复制和自动故障转移。本章后面部分将更详细地讨论集群复制。

容错一般与硬件相关，如备用电源、冗余的网络渠道等。对于软件系统，容错则是处理故障转移的附带结果。对 MySQL 集群来说，容错意味着允许一定数量的故障发生，且能够继续提供数据的访问。就像硬件 RAID 系统在同一个 RAID 阵列丢失两个驱动一样，replica 之间丢失多个数据节点会产生不可恢复的故障。但是，经过谨慎规划，可以配置 MySQL 集群以减小这种风险性。适当的监控和主动维护也可以减小风险。 549

MySQL 集群可以通过主动管理集群中的节点来达到容错。MySQL 集群使用心跳来检查服务是否可用，一旦发现节点故障，就会执行恢复。

MySQL 集群中的日志机制还提供了故障转移和容错的恢复级别。本地和全局检查点保证了集群中的数据是一致的。该信息对于数据节点故障的快速恢复非常重要，不仅允许数据的恢复，检查点的唯一性还允许节点的快速恢复。后面会详细讨论这个特性。

图 15-5 描绘了 Web 服务场景下 MySQL 集群的高可用性配置。

图 15-5 中的点线框表示系统边界。为了保证冗余这些组件都处于独立的硬件上，而且，要将四个数据节点配置为两个 replica。这幅图中没有显示与应用程序交互的附加组件，例如负载均衡器，用来分离 Web 和 MySQL 服务器的负载。

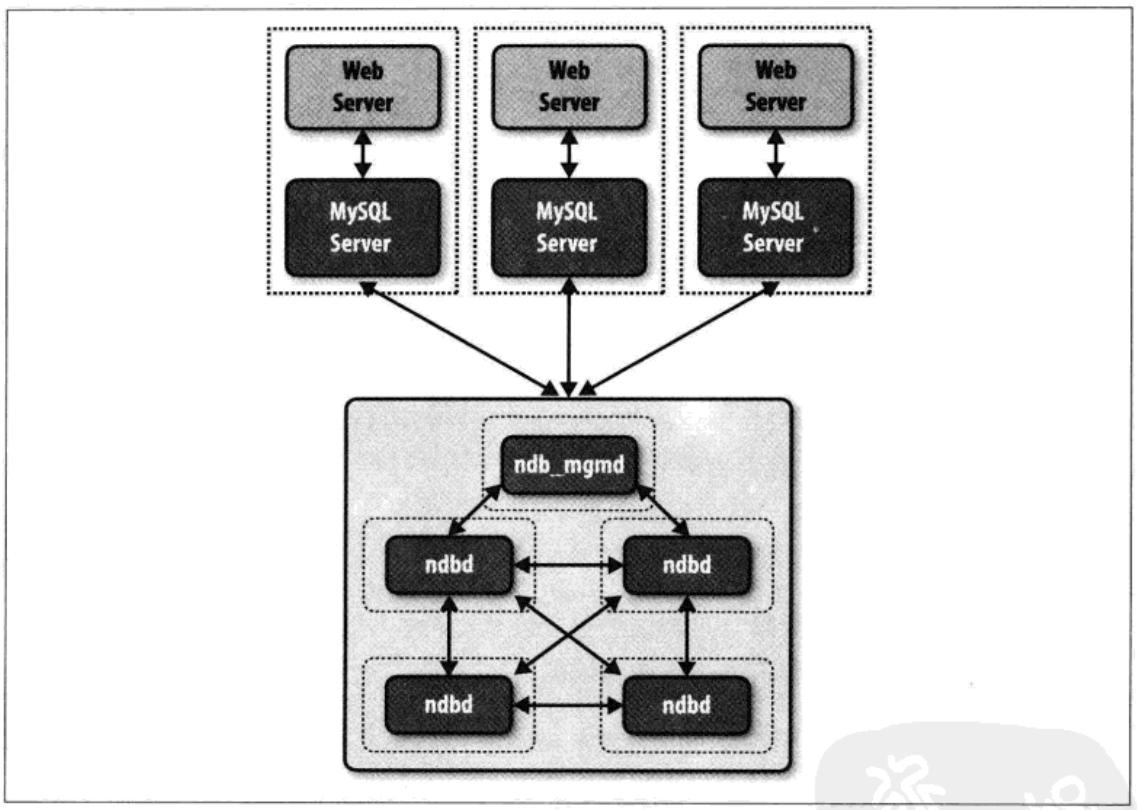


图15-5：高可用MySQL集群

550 在配置高可用性 MySQL 集群时，应该考虑使用下面的最佳实践。在后面考察高性能 MySQL 集群技术的时候，将进一步讨论这个问题。

- 在不同硬件的数据节点上使用多个 replica ；
- 使用冗余的网络连接以防止网络故障 ；

- 使用多个 SQL 节点；
- 使用多个数据节点来提高性能，并将数据分布化。

系统恢复

有两种类型的系统恢复，一种是维护或类似的计划性事件需要关闭服务器，另一种是非预期的系统性能丢失。幸好，MySQL 集群提供了一种恢复功能机制，即使最坏的情况发生。

如果 MySQL 集群被正确关闭，它会从日志中的检查点进行恢复。这主要是一个自动的常规启动阶段。系统会从每个数据节点的本地检查点装载最新数据，从而在重启时将数据恢复到最新的快照。一旦数据节点从本地检查点装载了数据，系统就会执行重做日志，以达到最近的全局检查点，从而将数据同步到系统关闭前的最后改变点。无论是有意关机后的重启，还是故障后的完全系统重启，这个过程都是一样的。

也许你认为启动是不能“恢复”的，但记住 MySQL 集群是一个内存数据库，因此，数据必须在启动时从磁盘重新装载。这个过程通过装载数据至最近的检查点来完成。

从重大灾难中恢复系统或进行纠正性措施时，还需要从数据的备份中恢复。前面提过，可以从 NDB 管理控制台调用 NDB_restore 实用程序，并使用最近的在线备份输出来恢复数据。

为了从备份中执行完整的系统恢复，首先要将集群设为单用户模式，在 NDB 管理控制台中使用下面的命令：

```
ENTER SINGLE USER MODE node-id
```

node-id 是指调用 NDB_restore 实用程序所在的数据节点的节点 ID。关于单用户模式和连接基于 API 的实用程序的更多详细信息，参见在线 MySQL 参考手册。

然后在集群的每个数据节点上运行恢复。一旦每个数据节点上的数据都恢复了，退出单用户模式，集群便可以使用了。要退出单用户模式，在 NDB 管理控制台中发出以下命令：

```
EXIT SINGLE USER MODE
```

551

关于 MySQL 集群的备份和恢复的更多信息，请参见在线 MySQL 参考手册中的“使用 MySQL 集群管理客户端来创建备份”和“恢复 MySQL 集群的备份”，具体链接如下：

http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-backup-using-management-client.html

http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-programs-NDB-restore.html



注意在故障或计划停机后重启服务器时不要使用 --initial 选项。

节点恢复

节点故障的原因有很多，包括网络、硬件、内存或操作系统的问题或故障。这里讨论这些故障最常见的原因，以及 MySQL 集群是如何处理节点恢复的。本节中重点考虑数据节点，因为它们是保证数据可访问的重要节点。

- 硬件
 当主机硬件故障时，显然运行在那个系统上的数据节点也会失效。这种情况下，MySQL 集群会故障转移到其他 replica 上。要对这种故障进行恢复，需要更换发生故障的硬件，然后重启数据节点。
- 网络
 如果数据节点的网络由于某种网络硬件或软件故障而变得不可用，节点或许可以继续执行，但由于它无法与其他节点取得联系（通过心跳），MySQL 集群会将该节点标记为“停止运行（down）”，然后故障转移到其他 replica，直到该节点恢复。这种故障的恢复需要更换发生故障的网络硬件，然后重启数据节点。
- 内存
 如果主机操作系统的内存不足，集群就会出现数据空间不足。这会导致数据节点失效。为了解决这个问题，可以增加更多的内存，或者增加配置参数的内存分配值，然后执行数据节点的滚动重启。
- 操作系统
 如果操作系统的配置干扰了数据节点的执行，就解决问题，然后重启数据节点。

552 关于数据库高可用性和 MySQL 高可用性的更多信息，参见下面的白皮书：

http://www.mysql.com/why-mysql/white-papers/mysql_db_high_availability.php
http://www.mysql.com/why-mysql/white-papers/mysql_ha_solutions.php

复制

我们已经简单讨论了 MySQL 复制及集群内部的复制之间的区别。MySQL cluster 复制有时又称内部集群复制（*internal cluster* 复制）或简称为内部复制（*internal* 复制），以表明它不是 MySQL 复制。MySQL 复制有时又称外部复制（*external* 复制）。

本节中将讨论 MySQL 集群内部复制，以及 MySQL 复制（外部复制）在 MySQL 集群之

间（而不是单个 MySQL 服务器之间）是如何进行数据复制的。

集群内部复制和MySQL复制

前面讲过 MySQL 集群内部使用同步复制，支持两阶段提交协议，保证数据完整性。相反，MySQL 复制使用异步复制，即依赖于稳定交付的单向的数据传输，而且在数据提交之前没有收到确认就开始执行。

集群内部的复制

内部 MySQL 集群复制通过存储数据的多个副本（称为 *replica*）来提供冗余。这个过程保证在查询被确认完成（提交）之前，数据被写入多个节点。这是通过两阶段提交实现的。

复制的形式是同步的，因为要保证在查询确认或提交完成时数据是一致的。

数据以片段（*fragment*）的形式复制，这里片段是指表中的记录行的一个子集。片段由于分区在各个数据节点之间分布开来，而且在每个 replica 的其他数据节点之上都有该片段的一份拷贝。其中一个片段将作为主拷贝，用来执行查询。所有其他的相同数据拷贝都作为备用片段。数据更新时，主片段会首先得到更新。

集群之间的MySQL复制

集群之间的复制很简单。如果你能够在两个 MySQL 服务器之间建立复制，那么就可以在两个 MySQL 集群之间建立复制。这是因为启动集群之间的复制并没有什么特殊的设置步骤或者额外的命令或参数。MySQL 复制就像单个服务器之间那样工作，数据存储 553
在 NDB 集群中。但是，外部复制有一些限制。这里列出了规划外部复制需要考虑的一些问题。如果想知道更多关于外部复制的信息，请参考在线 MySQL 参考手册的“MySQL 集群复制”一节。

- 外部复制必须是基于行的；
- 外部复制不能是环形的；
- 外部复制不支持 `auto_increment_*` 选项；
- 二进制日志的大小可能比常规的 MySQL 复制更大。

MySQL 复制将数据从一个集群复制到另一个集群，可以利用每个站点上 MySQL 集群的优势将数据复制到其他站点。

MySQL 复制可以在 MySQL 集群中使用吗？

可以从一个 MySQL 集群服务器复制到另一个非 MySQL 集群服务器（或反过来）。除了要考虑一些潜在的存储引擎冲突以外，不需要做任何特殊的配置，就像在使用不同存储引擎的 MySQL 服务器之间进行复制一样。这种情况下，也可以使用默认的存储引擎，而不用在 CREATE 语句中指定存储引擎。

从一个 MySQL 集群复制到一个非 MySQL 集群需要创建名为 *ndb_apply_status* 的特殊的表，来复制提交的时间。如果 Slave 上的这张表丢失，复制就会报错 *ndb_apply_status* 不存在而停止复制。可以使用如下命令创建这个表：

```
CREATE TABLE 'mysql'.'ndb_apply_status' (
  'server_id' INT(10) UNSIGNED NOT NULL,
  'epoch' BIGINT(20) UNSIGNED NOT NULL,
  'log_name' VARCHAR(255) CHARACTER SET latin1
    COLLATE latin1_bin NOT NULL,
  'start_pos' BIGINT(20) UNSIGNED NOT NULL,
  'end_pos' BIGINT(20) UNSIGNED NOT NULL,
  PRIMARY KEY ('server_id') USING HASH
) ENGINE=NDBCLUSTER DEFAULT CHARSET=latin1;
```

使用外部复制的 MySQL 集群复制需要基于行的 MySQL 复制，主 SQL 节点必须用 `--binlog-format=ROW` 或 `--binlog-format=MIXED` 启动。MySQL 复制的所有其他需求也都要满足（如所有 SQL 节点都有唯一的服务器 ID）。

外部复制还添加了一些特别的过程来支持集群之间的更新，包括使用集群二进制日志，binlog 注入线程和特殊的系统表来支持集群之间的更新等。外部复制在处理事务改变时也有所不同，这一点我们将在下一节中详细讨论。

554

MySQL集群（外部）复制的架构

可以认为外部复制与 MySQL 复制操作的基本概念是相同的。具体来说，某个集群安装时需要定义 Master 和 Slave，这样，Master 包含数据的原始拷贝，而 Slave 基于数据变化流增量地接收数据拷贝。

MySQL 集群复制使用 *mysql* 数据库中的专用表，它们位于 Master 和 Slave（无论 Slave 是单个服务器还是一个集群）的每个 SQL 节点上。这些表在 MySQL 安装过程中被创建。包括：*NDB_binlog_index* 表存储二进制日志（本地 SQL 节点的）的索引数据，*NDB_apply_status* 表存储已经复制到 Slave 上的操作记录。*NDB_apply_status* 表在所有 SQL 节点上维护，并保持同步，从而保证该表在整个集群中相同。

这些表由一个被称为 binlog 注射线程的新线程更新，这个线程通过在集群中记录变化来保证 NDB 集群存储引擎的任何改变都能在 Master 上更新。binlog 注射线程负责根据二进制日志中的记录来获取集群中所有的数据事件，并保证所有更改、插入或删除数据的事件都被写入 *NDB_binlog_index* 表中。Master 的转储（dump）线程使用 MySQL 复制将这些事件发给 Slave 的 I/O 线程。

外部复制与 MySQL 复制的一个重要不同之处在于，每个 epoch 都被视为一个事务。因为 epoch 是指每两个检查点之间的时间跨度，MySQL 集群保证每个检查点的一致性，所以 epoch 是原子性的，使用与 MySQL 复制中的事务相同的机制进行复制。关于最新应用的 epoch 信息存储在 NDB 系统表中，支持 MySQL 集群之间的外部复制。

单通道复制和多通道复制

Master 和 Slave 之间的 MySQL 复制连接称为一个通道（*channel*）。实际上通道是指 Master 连接 Slave 所使用的网络协议和方法。一般只有单通道，但为了保证最大可用性，可以建立一个备用通道用来容错。多通道外部复制如图 15-6 所示。

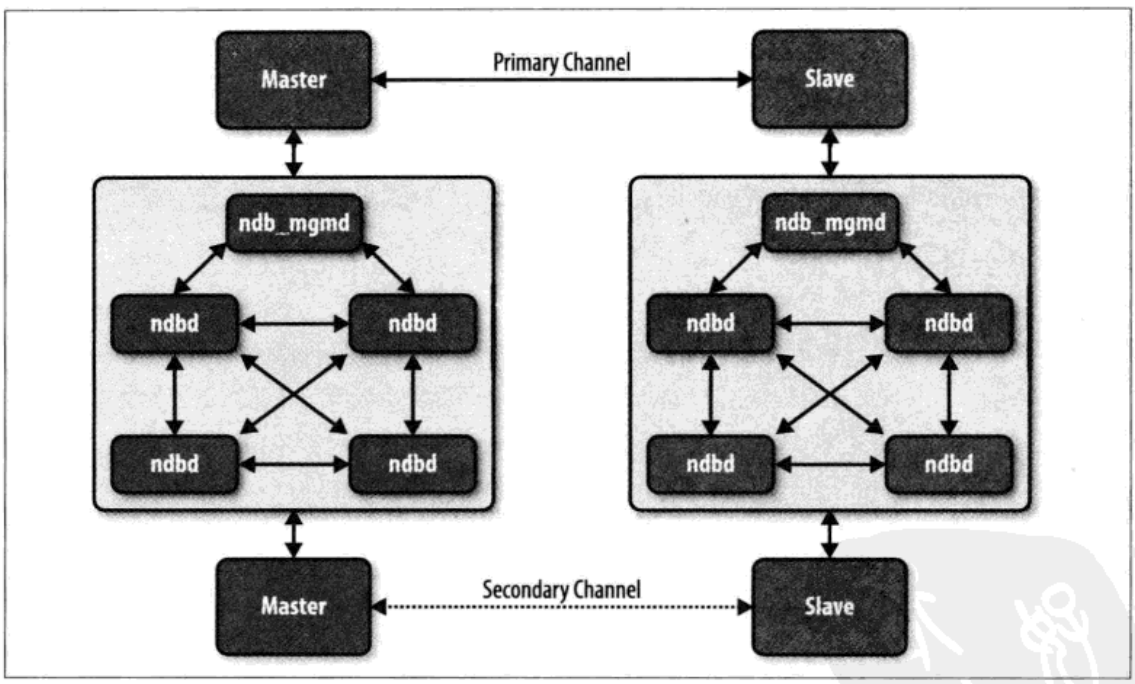


图15-6：多通道外部复制

多通道复制大大提高了网络连接故障的恢复能力。理想情况下，使用主动监控机制触发潜在的网络连接故障，以预示连接何时失效。该过程的实现方法很多，如采用简单心跳机制编写的脚本来警报和顾问，比如 MySQL Enterprise Monitor 中的脚本。

555 注意图 15-6 中的设置一共有 4 个 SQL 节点，作为 Master 的集群有 2 个 Master 节点，一个主（primary），一个备用（secondary）。同样，Slave 集群中也有一个主 Slave 和一个备用 Slave。主 Master/Slave 之间通过一个网络连接通信，备用 Master/Slave 使用另一个网络连接通信。



不要想当然认为网络组件是可靠的，即使一个交换器也会失效。同一个交换网络上使用不同布线起不了什么作用。最好是使用完全独立的冗余连接和中介网络组件，以达到真正的网络冗余。

多通道复制的建立与单通道（一般）MySQL 复制没什么不同。然而，复制的故障转移略有不同，其思想是无须在备用通道上启动 Slave。故障转移到备用通道上需要一些特别的步骤。

按照下面的步骤来启动多通道外部复制，激活主通道，使备用通道处于待机模式。假定冗余网络通信和硬件都已准备就绪并运行良好。

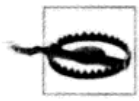
556 1. 启动主 Master；

2. 启动备用 Master；

3. 将主 Slave 连接到主 Master；

4. 将备用 Slave 连接到备用 Master；

5. 启动主 Slave。



不要启动备用 Slave，否则可能会造成主键冲突和数据重复的问题。但是，要将备用 Slave 连接到备用 Master，这样如果主通道发生故障，备用通道才能快速启动。

故障转移到备用通道的步骤则不同。仅仅启动备用 Slave 是不够的。为了避免同样的数据被复制两次，首先要确定最新复制的 epoch，然后用它启动复制过程。这个过程如下（注意使用变量保存中间结果）。



要保证主通道确实是离线的，为了以防万一还需要考虑停止主 Slave。

1. 找到 Slave 接收到的最近全局检查点的时间，这需要从主 Slave 的 *ndb_apply_status*

表中找到最近的 epoch。

```
SELECT @latest := MAX(epoch) FROM mysql.ndb_apply_status;
```

2. 获取故障后出现在主 Master 上 *ndb_binlog_index* 表中的行，可以用下面的查询从主 Master 中找到这些行：

```
SELECT @file := SUBSTRING_INDEX(File, '/', -1), @pos := Position
FROM mysql.ndb_binlog_index
WHERE epoch > @latest ORDER BY ASC LIMIT 1;
```

3. 同步备用通道。在备用 Slave 上运行下面的命令，这里 ***file*** 是实际文件名，***pos*** 是文件位置：

```
CHANGE MASTER TO MASTER_LOG_FILE = 'file', MASTER_LOG_POS = pos;
```

4. 在备用通道上启动复制，在备用 Slave 上运行这个命令：

```
START SLAVE;
```

这个故障转移过程会将故障转移到复制通道。如果 SQL 节点发生故障，则必须在执行故障转移之前处理并修复这些问题。

获得高性能

557

MySQL 集群并不仅仅是为了高可用性而设计的，还保证了高性能。我们已经回顾了很多高可用性的特性。本节中将考察提供高性能的一些特性。这里总结了一些获得系统高性能的最佳实践列表。

下面这些特性支持 MySQL 集群的高性能，其中有些前面章节已经考察过：

- 集群间复制（全局冗余）
 所有数据都复制到一个远程站点，这个远程站点可以卸载主站点。
- 集群内部复制（本地冗余）
 多个数据节点可以并行读取数据。
- 主存储器存储
 无须等待磁盘写，保证了数据更新的快速处理。

高性能的注意事项

使系统支持高性能有一些重要的注意事项。

- 保证应用程序尽可能高效。有时需要更改数据库服务器（如优化服务器配置或修改

数据库模式），而应用程序则通常可以被设计或重构成更高性能的应用。



Join 查询通常会很费时。

- 最大化数据库的访问。包括提供足够连接数（横向扩展）的 MySQL 服务器，以及为可用性而分发数据，如通过复制的方式。
- 考虑将你的 MySQL 集群性能提高，例如，增加更多的数据节点。

或许需要在期望的高可用性等级和高性能之间做出权衡。例如，通过添加更多的 replica 来增加可用性。但是，防止数据节点丢失的 replica 越多，就需要越强大的处理能力，而且更新过程的性能也会降低。数据读取依然很快，因为相同数据的读取并不需要多个 replica。保持较少的 replica 和较多的数据节点，可以获得更高的写性能。

558 另一个主要的注意事项是 MySQL 集群的分布式特性。因为每个节点在独立服务器上运行时效果最好，所以每个服务器的性能很重要，但网络组件也同样重要。由于协调性命令和数据在节点之间传输，网络的互联性能必须为高性能做调整。还要考虑一些参数，如端口选择（如 TCP/IP, SHM, SCI）、延迟、带宽和地理上的接近度。



可以在云环境中搭建和运行 MySQL 集群。这样做的好处之一就是网络互联速度很快，而且是优化过的。由于数据节点需要快速的处理器、足够的内存和快速的网络，所以在云环境中使用 MySQL 集群使用虚拟服务器技术绰绰有余。

从在线 MySQL 参考手册的“MySQL 集群”一节中可以找到完整的注意事项列表。关于 MySQL 性能提升，可以参见 Baron Schwartz 等人的《高性能 MySQL》一书（O'Reilly）。

高性能的最佳实践

保证 MySQL 集群运行的性能最高有很多办法。这里列出了一些最常用的性能提升实践，每项还附有简要的讨论。其中有些较为常见，但我们并不想忽略可能达到高性能需求的所有特性。

- 调整访问模式

 考虑应用程序访问数据的方法。由于 MySQL 集群在内存中存储索引字段，所以在非索引字段上引用这些列来访问，比单个 MySQL 服务器速度快得多。MySQL 集群每张表都有主键，所以根据主键检索数据的应用程序大部分都很快。

- 使你的应用程序是分布感知的（*distribution-aware*）
 在已分区的数据存储上访问数据最好将集群中单个节点的访问区分开来。默认情况下，MySQL 集群使用主键将行哈希到各个分区。但不幸的是，如果考虑到主表 / 从表（master/detail）的查询（由从表查询主表，且该从表参照了主表），这样做并不总是最佳的。这种情况下，就要修改哈希函数，保证主表中的行和从表中的行位于同一个节点上。一种实现方式是分区裁剪（*partition pruning*），将从表哈希分区时使用的字段删除，仅根据主表的主键（即详细表的外键）对从表中的行进行分区。559
 这样主表和详细表中的行就被分配到分区树（partition tree）中的同一个节点上了。
- 使用批量操作
 每个查询的往返都具有显著的开销。像插入这样的操作，可以使用多条插入查询（一次插入多行的 INSERT 语句）来减少开销。打开 transaction_allow_batching 参数，并在单个事务（BEGIN 和 END 之间的语句块）中包含多个操作，可以进行批量操作。这样你可以列出多个数据操作查询（INSERT，UPDATE 等），并减少开销。



transaction_allow_batching 选项不能在 SELECT 语句和带变量的 UPDATE 语句中使用。

- 优化数据库模式
 MySQL 集群中的数据库模式优化与一般的数据系统一样。MySQL 集群使用高效的数据类型（如节省内存所需的最小规模；一张百万行记录的表每行 30 字节，可以节省大量的内存）。为了利用 MySQL 集群的并行数据访问方法（分区），还要考虑将某些模式非规范化（denormalization）。
- 优化查询
 显然，查询被优化得越多，查询性能就越高。对所有数据库来说都是这样，而首先要做的就是提高应用程序的性能。对 MySQL 集群来说，从数据检索的角度考虑查询优化。MySQL 集群的性能对 join 操作尤其敏感。性能很差的查询有时可能导致异常，容易被误认为是系统其他部分的低效所致。
- 优化服务器参数
 优化集群配置可以保证其运行尽可能地高效。这意味着要花些时间理解很多配置选项，而且要确定使用正确的硬件。这项任务没有任何神奇的药水——参数修改得越多，你的安装设置就越独一无二。这项实践要谨慎使用，一次只修改一个参数，并且总要在改变之前与已知的基准进行比较。
- 使用连接池
 默认情况下，SQL 节点仅使用单个线程连接 NDB 集群。如果有更多的连接线程，

SQL 节点可以一次执行多个查询。要想在 SQL 节点中使用连接池，需要向配置文件中添加 NDB-cluster-connection-pool 选项。将值设为大于 1 的数（比如 4），并把这个选项放在 [mysqld] 段中。应该用这个设置进行试验，因为对于应用或硬件来说，这个值通常太高了。

560

- 使用多线程数据节点

 如果数据节点有多核 CPU 或多个 CPU，运行名为 NDBmtd 的多线程数据节点守护进程，可以获得额外的性能增加。这个守护进程使用高达 8 个 CPU 核或线程。多个线程使得数据节点可以并行执行多个操作，如本地查询处理器（Local Query Handler，LQH）和通信进程，以获取更高的吞吐量。

- 为自定义应用程序使用 NDB API

 MySQL 服务器（即 SQL 节点）提供了一个快速的查询处理器前端，而 MySQL 构建了一种直接访问 C++ 的机制，称为 NDB API。对某些操作来说，如 MySQL 集群与 LDAP 的接口，这可能是将 MySQL 集群（这里指的是 NDB 集群）连接到你的应用程序的唯一办法。如果应用程序的性能要求很高，而且你具备自定义 NDB API 方案所需的开发资源，你会发现性能将得到大大提升。

- 使用正确的硬件

 一般硬件越快就会导致性能越好。但是，要考虑到集群配置的各个方面。不仅要有更快的 CPU 和更快的内存，还要有诸如 SCI 之类的高速网络互联，以及高速的硬件冗余的网络连接。很多情况下，这些硬件方案被构建成可立即投入使用的商品，而不用重新配置集群。

- 关闭查询缓存

 因为 MySQL 集群不使用 MyISAM 存储引擎，所以查询缓存没有什么用处，可以关闭它。

- 不要使用交换空间

 确保你的数据使用的是真正的内存而不是交换空间。如果数据节点使用交换空间启动的话，性能会急剧下降。这不仅仅是性能问题，还可能影响集群的稳定性。

- 数据节点使用处理器亲和度

 在多 CPU 机器中，锁定数据节点的 CPU 处理不受网络通信影响。在某些平台（如 Sun CMT 处理器系统）上修改配置文件可以达到这个目的，即在 [NDBd] 段使用 LockExecuteThreadToCPU 或 LockMaintThreadsToCPU 参数。

如果遵从上述最佳实践，就可以使 MySQL 集群达到最佳性能和最高可用性。想了解更多关于优化 MySQL 集群的信息，请参见白皮书“MySQL 集群数据库的性能优化”，网址为 *http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster_perfomance.php*。

小结

本章讨论了MySQL集群中MySQL独特的高可用性方案。MySQL集群的强大之处在于将表分区，然后将它们分布到独立的节点上去，MySQL集群的并行架构与多主（multimaster）数据库相当。这样系统就可以并发地执行大量的读写操作。所有的更新对所有存储在数据节点上的应用程序节点（通过SQL命令或NDB API）立即可用。

因为写负载被分布到所有数据节点上，你可以获得很高的写吞吐量，以及事务性工作量的可扩展性。最后，多个MySQL服务器节点并行运行，其中每个服务器通过多个连接共享负载，并使用MySQL复制确保不同地理位置站点之间的数据传输，你就可以构建高性能、高并发的事务性应用程序。

MySQL集群对那些要求最大限度的MySQL高可用性的应用程序是一个很好的解决方案——尽管极少数应用程序有这样严格的需求。

"Joel！"

老板折回来站到门边，Joel笑道，"什么事，Bob?"

Summerson先生走进办公室关上门，然后拉过一把椅子，坐在Joel对面。

顷刻之间Joel措手不及，只是笑着说，"我可以为你做点什么吗，Bob？"

"你已经帮我做过了，Joel。你快速解决了MySQL的最新情况，也跟上了我们加快发展和近期收购的步伐，现在你又帮我在这笔买卖中赚了很多钱。我知道我给你安排了很多任务，你也应该得到一些回报了。"一个令人不安的停顿之后，他问道，"你打高尔夫球吗，Joel？"

Joel耸了耸肩，"大学以后就不玩了，而且我也向来打不好。"

"没关系。我喜欢这个游戏，但感觉不是相互的，我每次打球都会丢掉一半的球。周六有时间玩9个洞吗？"

Joel不知道哪里出了问题，但直觉告诉他应该接受这个请求。"当然，我有时间。"

"很好，我们上午10点在Fair Oaks碰面。我们先玩9个洞，然后午饭的时候讨论一下你的将来。"

"好的。到时候见，Bob。"

562

Summerson 先生站起来，打开门，停了下来，“我已经让会计处编制一个由你管理的预算，包括 MySQL 企业版订购费，和雇佣两个全职助手的费用。”

“谢谢。” Joel 感动地说。他还没有准备好完全接受他的提议，更不用说还有更多的职责。

Summerson 先生消失在大厅后，Joel 的朋友 Amy 走进来站到他旁边，“你还好吧？”她关心地问。

“好啊，怎么了？”

“我以前从没见他关门跟别人谈话的。如果你不介意，你们都说什么了？”

Joel 挥了挥手，说：“他叫我去玩高尔夫，然后说我有自己的预算了，可以订购 MySQL 企业版。”

Amy 笑了，抓住他的胳膊，“那就好，Joel，真好。”

Joel 困惑了，他觉得负责管钱或者同意购买订单不至于有这么大反应。“怎么了？”

“上一个和 Summerson 先生打高尔夫的人升职加薪了。Summerson 看起来很无情，其实还蛮值得为他卖命的。”

“真的吗？” Joel 盯着桌上的文件，告诉自己不要期望太高。

“中午一起吃饭吗？” Amy 轻轻挤了一下他的胳膊问道。

Joel 看了看 Amy 放在自己胳膊上的手笑着说，“好啊，我们去个好地方吧。”但在接受了她的要求之后，Joel 知道可能要为了下一次约会而熬夜工作。

复制建议和窍门

下面介绍关于运行、诊断、修复和改善 MySQL 复制的有用建议和窍门。

由于这一章只是附加的知识，所以可能不会包括它们的具体细节。可以从在线 MySQL 参考手册 *http://dev.mysql.com/doc/refman/5.4/en/index.html* 找到有关 MySQL 复制的更加详细的内容。

同样可以在复制开发小组 *http://dev.mysql.com/replication/* 找到最新的、最先进的技术。

附录的最后几个小部分描述了即将在 MySQL 上提供的特性，但在写这本书的时候官方还不能提供这些特性。

Slave停机了，怎么办

如果你的 Slave 在没有发出一个警告和错误信息的情况下停止运行，这时你应该在文档中查找错误发生的可能原因，并执行任何必需的修复。

一旦你校正了错误，在发生错误后按照以下步骤确定开始从哪里着手重启 Slave。

1. 使用以下命令检查停机前的位置：

```
SHOW SLAVE STATUS
```

2. 使用 `Master_Log_File` 和 `Read_Master_Log_Pos` 参数来确定下一个从 Master 传过来的事件。

3. 使用 `Relay_Master_Log_File` 和 `Exec_Master_Log_Pos` 参数来确定下一个 master

log 请求事件。

4. 使用 Relay_Log_File 和 Relay_Log_Pos 参数来确定下一个 relay log 请求事件。

564 5. 使用 mysqlbinlog 来读取以下内容：

```
mysqlbinlog master-log.000001
mysqlbinlog relay-log.000001
```

6. 调查这个问题，并且如果需要，在数据库中删除几行。

```
SET SQL_SLAVE_SKIP_COUNTER=1; START SLAVE
```

检查冗长的二进制日志

如果你使用基于行的日志方式，可以使用 --verbose 选项查看在事件中重建的查询。以下二进制日志运行在基于行的日志方式

```
$ mysqlbinlog --verbose master-bin.000001
BINLOG '
qZnvSRMBAAAAKQAAAAYCAAAAABAAAAAAAAAAAAAABHRlc3QAAnQxAAEDAAE=
qZnvSRcBAAAAJwAAAC0CAAAQABAAAAAAAAAEAAf/+AwAAAP4EAAAA '/*!*/;
### INSERT INTO test.t1
### SET
### @1=3
### INSERT INTO test.t1
### SET
### @1=4
```

注意，列是以 @n-style 方式进行命名的，这是因为基于行的复制时，传输和应用都是以列的位置而不是以列名来定义数据的。

利用复制在表中重建数据

如果 Slave 上的表由于错误或事故（如一个用户删除了数据）而被损坏，可以使用复制来恢复这个表的数据。首先在 Master 上创建一个原始表的临时副本，然后删除原始表，最后使用临时表的数据重新创建原始表。只要不是有些特殊的列数据类型（如 autoincrement）会影响到结果，这种方法都会可行。使用复制在 Slave 上重建表。按照日志的方式不同，重建的方式也有所不同。

565 ## 基于语句的日志

如果你使用基于语句的记录日志的方式，在每张表上运行下述语句。

```
SELECT * INTO OUTFILE 't1.txt' FROM t1;
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 ...;
LOAD DATA INFILE 't1.txt' INTO TABLE t1;
```

基于行的日志

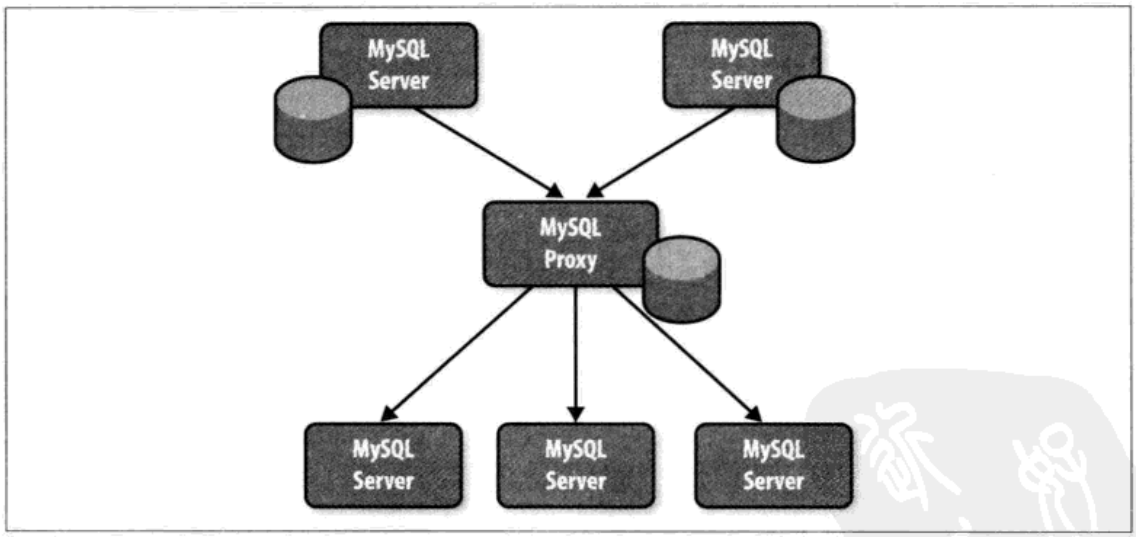
如果你使用基于行的记录日志的方式，临时表不会被传送到 Slave。因此，只能将所有数据用 INSERT INTO 命令建表，方法如下：

```
CREATE TEMPORARY TABLE t1_tmp LIKE t1;
INSERT INTO t1_tmp SELECT * FROM t1;
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 SELECT * FROM t1_tmp;
```

使用MySQL Proxy来完成多Master的复制

Slave 不能拥有一个以上的 Master，Master 却可以拥有多个 Slave。在很多安装环境下这都不是问题，但是如何合并两台不同的 Master 数据并同时复制这个组合数据到 Slave 上呢？

一种方法是使用 MySQL Proxy 作为 Master 和 Slave 的中介，图 A-1 展现了这种多 Master 的配置概念图。



图A-1：使用MySQL Proxy来处理多Master

MySQL Proxy 从两个 Master 接收改变（事件），并写入一个新的二进制日志中，但不需要保存数据（写入数据库）。你的 Slave 将 MySQL Proxy 当做 Master 使用。来自两台 Master 的组合数据就可以复制到一套 Slave 上去了。这可是一个有效的多 Master 复制的

实现。

如果需要了解更多有关 MySQL Proxy，请参考 MySQL Proxy 在线参考手册：*http://dev.mysql.com/doc/refman/5.4/en/*。

566

使用默认的存储引擎

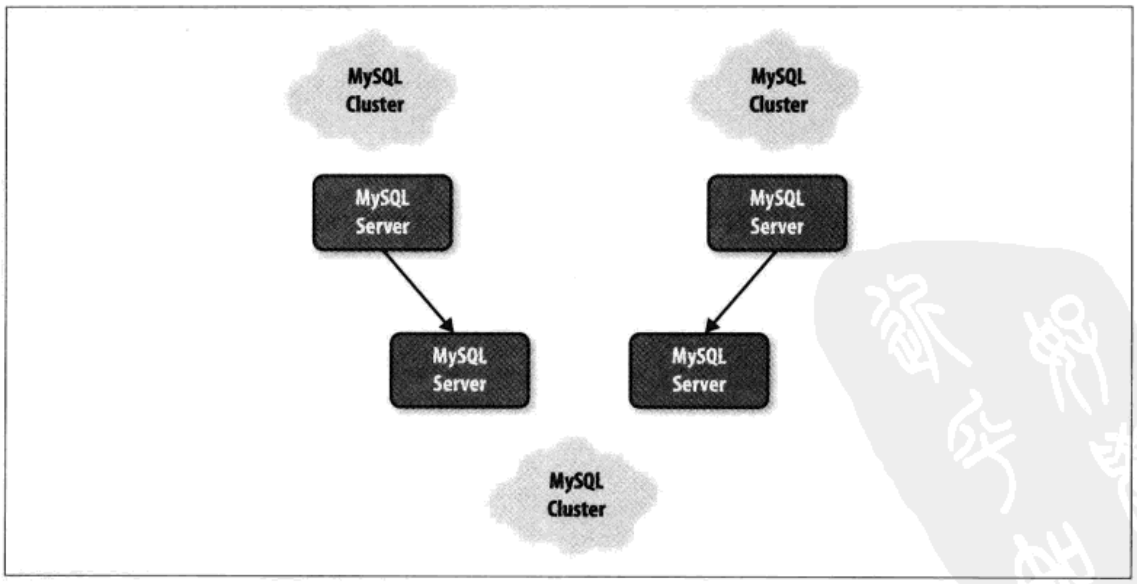
如果你在 Master 上使用 `SET GLOBAL STORAGE ENGINE` 命令设置默认的存储引擎，这个命令是不会被复制的。因此，任何没有指定存储引擎的 `CREATE` 语句将会使用 Slave 上的默认存储引擎。因此，有 InnoDB 表复制到 MyISAM 表的可能性，当这种情况发生时而你又忽略警告，就可能会因为某些不兼容而导致复制中断。

一种解决方案是确保在每台 Slave 上使用 `default-storage-engine` 选项设置全局存储引擎。然而最佳实践是在 `CREATE` 语句上指定存储引擎。

如果 Master 使用存储引擎，但在 Slave 上却没有，同样会导致错误的发生。这种情况下只能在 Slave 上安装缺失的存储引擎。

MySQL Cluster 多源（Multisource）复制

你可以使用 MySQL Cluster 来设置多数据源复制（如图 A-2 所示）。为达到这个目的可以使用多个 Master 复制不同的数据。使用不同的 Master 存储数据可以避免主键和一些上下文相关的值的交叉污染。



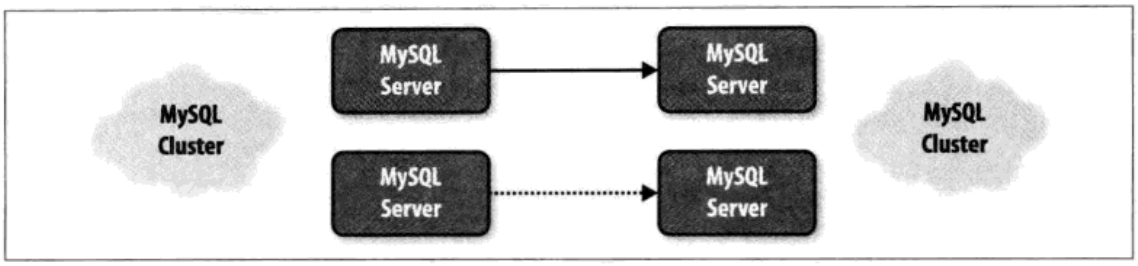
图A-2：多源复制

多路（Multichannel）复制故障转移

567

为了更好的可用性和快速恢复，可以使用双复制（如图 A-3 所示），在双复制中两套复制拓扑结构从一个 Cluster 复制到另一个。如果一条数据流发生了失败，可以切换到另一条数据流上去。

注意，服务器须以 *mysql.X* 方式命名，保存位置以 *mysql.X.savepos* 方式命名。



图A-3：多路复制

使用当前数据库来过滤

binlog 过滤器可以很快、很方便地为特定目的过滤不要的语句。

例如，有些语句只是在本地服务器上执行，如设置数据表的存储引擎，其实你并不希望 Slave 上的存储引擎同 Master 上一样。有很多原因希望 Master 和 Slave 上采用不同引擎，如下：

- MyISAM 更适用于报告，因为二进制日志包含很复杂的事务，可以使用 Master 上的 InnoDB 执行事务进程，使用 Slave 上的 MyISAM 生成报告和提供分析处理。
- 为了节约内存，可以将分析处理服务的 Slave 上的某些表设置为 Blackhole 存储引擎。

可以将 `SQL_LOG_BIN` 服务器参数设置为 0，从而将二进制日志写入过程悬挂起来。如例 A-1 所示，在将表引擎改成 MyISAM 前先关闭日志。因此，这次改变只会在 Master 上起作用，改完后再将日志开启。然而，修改 `SQL_LOG_BIN` 需要 `SUPER` 权限，但是你又不想将权限赋予普通用户。因此，例 5-4 给出了另一个替代方法来改变 Master 上存储引擎而不会被日志记录下来（在例子中，假设 `my_table` 表在 `my_db` 数据库中）。创建一个专用数据库（在本例中叫 `no_write_db`）。此时，任何用户在 `no_write_db` 上执行的语句都不会被复制，这些语句将会被直接过滤掉。因为使用数据库需要连接权限，你可以控制谁拥有权限来隐藏语句，而不需要将 `super` 权限赋予不信任用户。

568

例 A-1：两种从二进制日志过滤语句的不同的方式

```
master> SET SQL_LOG_BIN = 0;
Query OK, 0 rows affected (0.00 sec)

master> ALTER TABLE my_table ENGINE=MyISAM;
Query OK, 92 row affected (0.90 sec)
Records: 92 Duplicates: 0 Warnings: 0

master> SET SQL_LOG_BIN = 1;
Query OK, 0 rows affected (0.00 sec)

master> USE no_write_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
master> ALTER TABLE my_db.my_table ENGINE=MyISAM;
Query OK, 92 row affected (0.90 sec)
Records: 92 Duplicates: 0 Warnings: 0
```

Slave上的列比Master上多

如果你需要在 Slave 上增加列，但不想改变 Master（例如记录时间戳或者增加路径或者其他本地化数据），你可以在 Slave 的表上增加列而不需要对 Master 做任何改变。MySQL 复制支持这种更多列的场景。为了在 Slave 额外的表中插入数据，可以定义它们能够接受的默认值（如插入 timestamps 就比较方便），或者在 Slave 上定义 trigger 来为它们提供数据。

对于基于语句的记录日志方式，可以使用以下方法创建列：

1. 在 Master 上创建表：

```
CREATE TABLE t1 (a INT, b INT);
```

2. 在 Slave 上更改表：

```
ALTER TABLE t1 ADD ts TIMESTAMP;
```

3. 在 Master 上插入值的例子：

```
INSERT INTO t1(a,b) VALUES (10,20);
```

对于基于行的记录日志方式，新列必须出现在行的最后，同时包含默认值。如果在行的中间或者开头创建新列，基于行的复制就会出错。只要在行的末尾增加新列并给予默认值，就不需特别需要考虑哪些语句需要被复制，因为基于行的复制需要直接从行中抽取列来做更改、添加和删除操作。

569

Slave上的列比Master上少

如果需要 Slave 上的列比 Master 上更少，例如，保护一些敏感数据，或减少数据传输量，可以在 Slave 上减少列而不对 Master 做任何改变。MySQL 基于行的复制支持这样的缺少列的场景。然而，缺少的列必须是 Master 行中最后的几列。

在基于行复制中，可以用以下方法删除列。

1. 在 Master 上创建表：

```
CREATE TABLE t1 (a INT, b INT, comments TEXT);
```

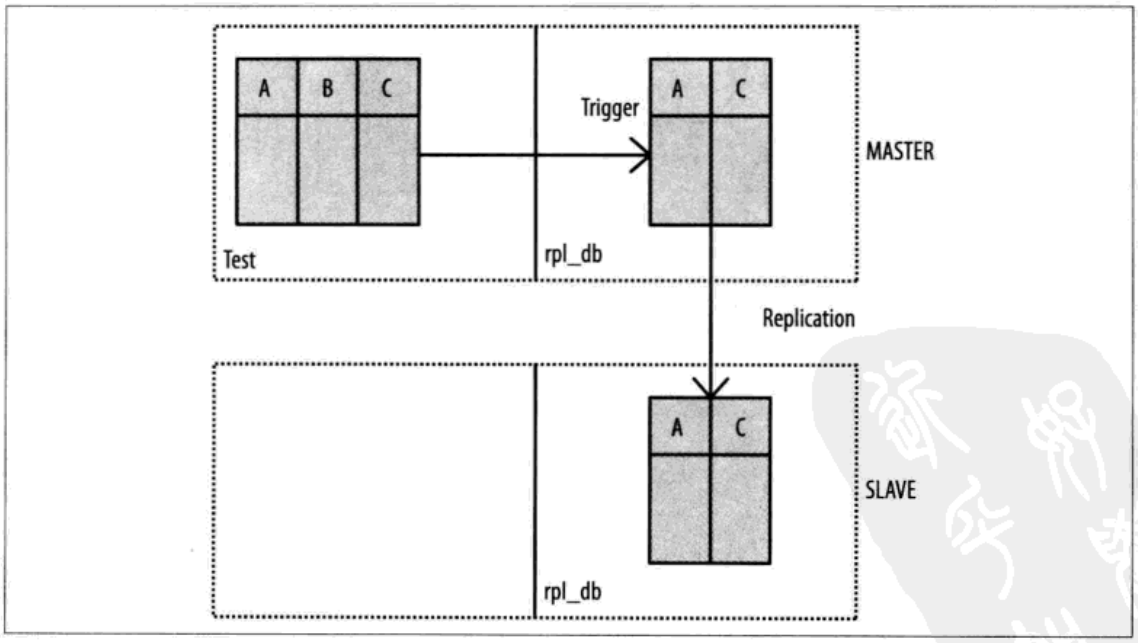
2. 在 Slave 上更改表：

```
ALTER TABLE t1 DROP comments;
```

3. 在 Master 上插入值的例子：

```
INSERT INTO t1 VALUES (1,2,"Do not store this on slave");
```

对于基于行的复制，可以复制任意一组列，而不用考虑它们是否是表的最后一列，或者也不用考虑它们是否在 Master 上使用 trigger 将另一个数据库中基础表的数据更新到新表中，在复制的时候只是复制新表而不是基表。图 A-4 所示为方案概念图，这里使用 trigger 方式将一个有 3 个列的表复制到 Master 本地另一表中。



图A-4：复制列子集

570 下面的例子给出了在被复制数据库中如何使用 trigger 更新表。在 Master 上，执行步骤如下：

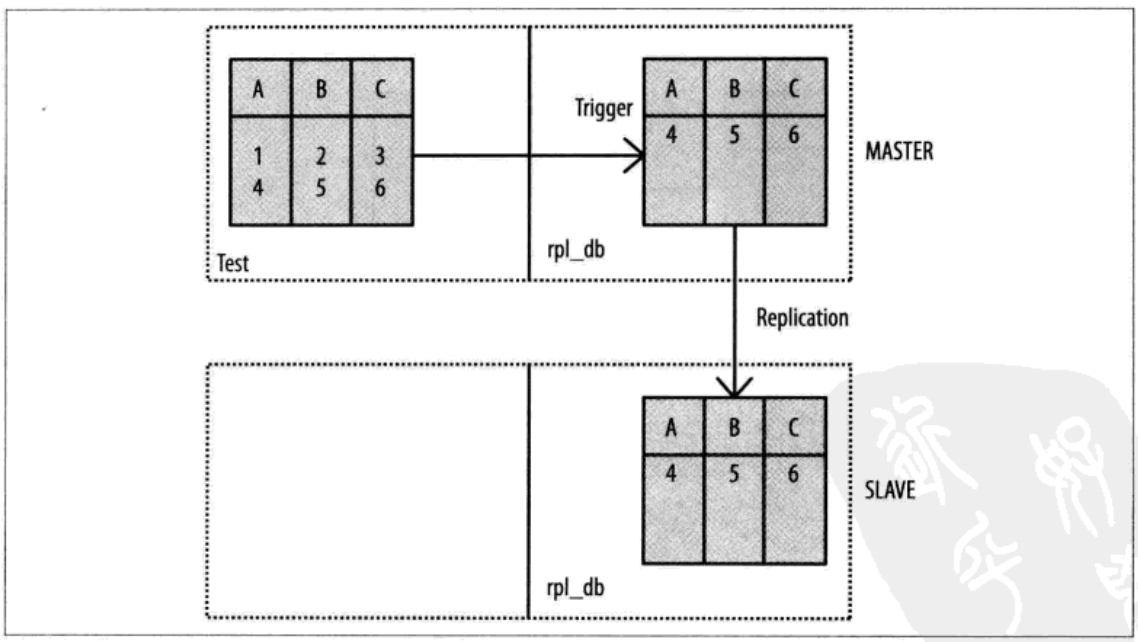
```
CREATE DATABASE rpl_db;
USE test;
CREATE TABLE t1 (a INT, b BLOB, c INT);
CREATE TRIGGER tr_t1 AFTER INSERT ON test.t1 FOR EACH ROW
INSERT INTO rpl_db.t1_v(a,c) VALUES(NEW.a,NEW.c);
USE rpl_db;
CREATE TABLE t1_v (a INT, c INT);
```

当执行一个正常的 insert 语句时，trigger 将会抽取语句中列的子集，然后将数据写入一个新加的 *rpl_db* 数据库表中。然后就从该数据库中复制数据。

```
USE test;
SET @blob = REPEAT('beef',100);
INSERT INTO t1 VALUES (1,@blob,3), (2,@blob,9);
```

选择某几列复制到Slave

也可以只复制部分符合要求的行。这个技术使用一个从 Master 复制到 Slave 的专用数据库（可以用过滤器来实现），并使用 trigger 来判断哪些行将会被复制，并将需要复制的行插入到复制表中。



图A-5：复制行子集

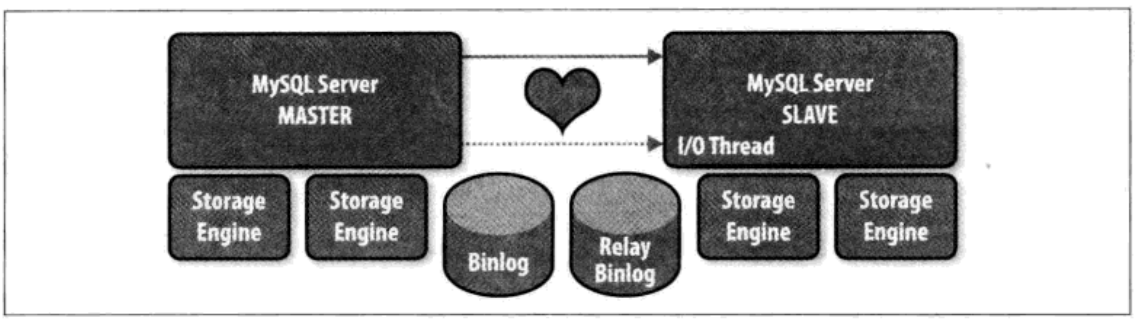
下面的例子给出了在被复制数据库中插入数据的时候，如何使用 trigger 更新那些只有一个奇数行的表：

571

```
# Just replicate rows that has odd numbers in first column.
USE rpl_db;
CREATE TABLE t1_h (a INT, b BLOB, c INT);
--delimiter //
CREATE TRIGGER slice_t1_horiz AFTER INSERT ON test.t1
FOR EACH ROW
BEGIN
  IF NEW.a MOD 2 = 1 THEN INSERT
    INTO rpl_db.t1_h VALUES (NEW.a, NEW.b, NEW.c);
  END IF;
END//
--delimiter ;
```

复制心跳

使用复制心跳机制可以提高复制的可用性和故障转移速度。当心跳开启的时候，Slave 周期性地询问 Master 状态。如果 Slave 获得响应，它会在下一个周期后继续询问 Master。服务器维护着一组收到心跳的统计数字，这些数字可以帮助诊断 Master 是离线状态还是失去响应。图 A-6 给出了心跳机制的概念图。



图A-6：复制中的心跳

心跳机制使得 Slave 发现何时 Master 可用，这有以下几点好处和能力：

- 自动检查连接状态；
- 当 Master 空闲时 relay log 不轮换；
- 诊断 Master/Slave 失去连接可配置在毫秒级别。

因此，可以在实施自动故障转移和类似高可用性操作时使用心跳机制。在 Slave 上使用以下命令来设置心跳的间隔时间：

```
CHANGE MASTER SET master_heartbeat_period= val;
```

572 使用下列命令来检查心跳机制的设置和统计信息：

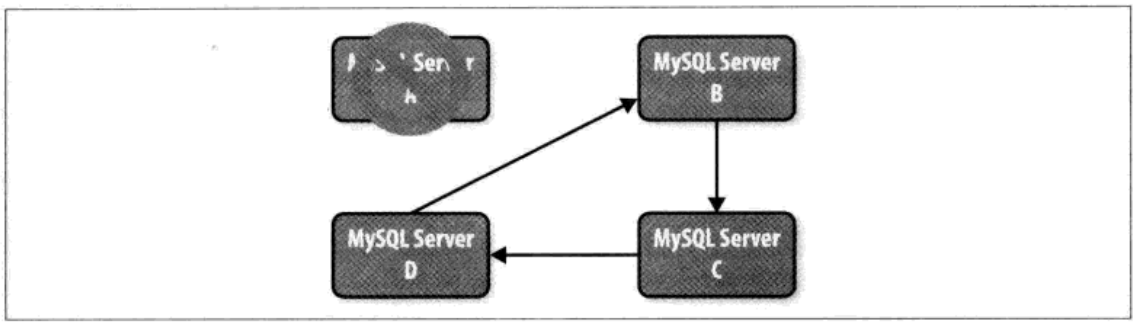
```
SHOW STATUS like 'slave_heartbeat period';
SHOW STATUS like 'slave_received_heartbeats';
```



这个特性相对来说较新。它是在 MySQL 5.4.4 版本中引进的。

在环形复制中忽略服务器

在一个环形复制拓扑中，当一个服务器失败时，必须将这个失败的服务器从拓扑中删除。在图 A-7 中，服务器 A 失败了，需要将其从环中删除。在本例中，设置服务器 B 来终止服务器 A 在新环中的事件。



图A-7：在环形复制中替换服务器

可以在服务器 B 上设置如下命令：

```
CHANGE MASTER TO MASTER_HOST=C ... IGNORE_SERVER_IDS=(A)
```

这个特性是在 MySQL 5.5 版本中引进的。

功能预览：延时复制

有的时候需要延时复制，如确认 Slave 对于更改不会过载。该特性使得 Slave 能够总是在执行复制的时候比 Master 要慢 *n* 秒。你可以设置下列参数：

```
CHANGE MASTER TO MASTER_DELAY= n
```

n 是一个小于 MAX_ULONG 的非负整型数值，用于设置 Slave 需要等多少秒（如果这个值

强行被设置得更高，会被拒绝并报错）。

可以通过 SHOW SLAVE STATUS 命令来检查延时安装情况和检查 Seconds_behind_master 列的输出。 <573>

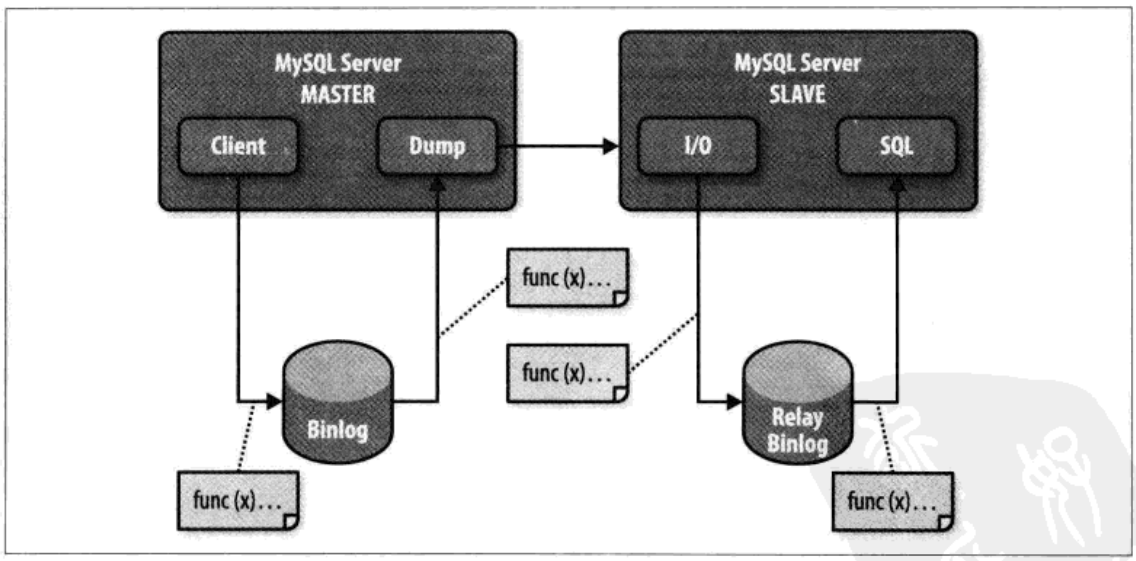
可以在 *http://forge.mysql.com/wiki/ReplicationFeatures/DelayedReplication* 了解到更多的有关延时复制的信息。

功能预览：脚本式复制

如果你需要执行转换，指定低等级的过滤器，或者在复制流上做些操作，在下列四个时期运行脚本：

- 当事件被写入 Master 上的二进制日志时；
- 当事件被写入 Master 上的 dump 线程时；
- 在 I/O 线程读到事件后，但在写入 Slave 的 relay log 之前；
- 当 Slave 从 relay log 上读取事件时。

依赖于你的需求，你可能希望在其中一个或多个执行点上执行一些操作。图 A-8 显示了脚本点的位置。此功能允许你使用 Lua 语言执行脚本。



图A-8：可以插入脚本的点

你可以在 *http://forge.mysql.com/wiki/ReplicationFeatures/ScriptableReplication* 上查阅到更多的关于脚本式复制。

574 功能预览：Oracle算法

作为脚本复制解决方案的一部分，你可以使用 Paul Tuckfield 开发的“Oracle”算法（谷歌 /YouTube）来过滤事件（如图 A-9 所示）。

```
module(..., package.seeall); require "luasql.mysql"
pattern = {
        [ "UPDATE%s+(%w+) .*%s(WHERE.*)"] = "SELECT * FROM %1 %2",
        ["DELETE%s+FROM%s+(%w+) .*%s(WHERE.*)"] = "SELECT * FROM %1 %2",
}
env = luasql.mysql ()
con = env:connect("test", "root", "", "localhost", mysql.port)

function before_write(event)
        local line = event.query
        if not line then return end
        for pat,repl in pairs(pattern) do
                local str = string.gsub(line, pat, repl)
                if str then con:execute(str) ; break; end
        end
end
```

图A-9：Paul Tuckfield 的Oracle算法

有趣的是，该算法可以让复制线程“预言”未来的事件和事先准备相应的缓存，并因此而得名，而与 Oracle 公司或 Oracle 的 text-scoring 算法毫无关系。



如果你使用 Maatkit，可以使用 `mk-slave-prefetch` 来达到与使用 SQL 命令相同的功能。可以从 *http://www.maatkit.org/doc/mk-slave-prefetch.html* 了解更详细的信息。

索引†

符号

*（星号），92
#（井号），90，
_（下画线），164

A

当前 binlog 文件
 定义，20，47
 切换，83
主动内容，488
Activity Monitor，273-276
管理性任务
 管理复制，23-25，153
 执行常见的任务，36-43
顾问
 定义，453
 企业仪表盘，468，
 支持 MEM，457
代理
 定义，453
 修复问题，462
 支持 MEM，457
Akamai（供应商），486
Alexa 网站排名服务，489
Alexa Web 信息服务，489
ALTER 语句
 Binlog 事件，48
 记录事务，76
ALTER TABLE 语句
 级联复制，160
 隐式提交，135
 记录日志，51
 Myisamchk 实用工具，347
 MySQL 集群，538
 故障排除查询，394
 使用示例，334
Amazon（供应商），486
Amazon CloudFront CMS，488，499
Amazon CloudWatch，488
Amazon DevPay 服务，489

†：索引部分给出的页码为英文版页码，请参照用“⇨”给出的原书页码。

Amazon 弹性块存储（EBS），489，492，511–516
Amazon 弹性计算云（EC2）
Amazon 管理控制台，493
AMI 工具，483，496，508–511
API 工具，483，495，515–516
最佳实践，520-522
浏览器插件，494
命令行工具，495，498
弹性，487
功能，488，489
复制，517–520
运行实例，501
Amazon 实现 Web 服务（FWS），488
Amazon 机器映像（AMI），483，492，501
Amazon 土耳其机器人，489
Amazon 关系数据库服务（RDS），488
Amazon 简单队列服务（SQS），488
Amazon 简单存储服务（S3），488，491
Amazon SimpleDB，488
Amazon 虚拟私有云（VPC），489
Amazon Web Services（AWS）
AMI（Amazon 机器映像），483，492，501
ANALYZE TABLE 命令，328–329，339，345
Append_block 事件，57
归档计划
定义，423
形成，423–424
Archive 存储引擎，335
星号（*），92
异步复制
MySQL 支持，6
意义，150–152
自动伸缩，488
自动备份，446–448
AUTO_INCREMENT 属性
最佳实践，338
双向复制，123
上下文事件，53
记录查询，52
使用示例，218
auto_increment_increment 选项，123，399
auto_increment_offset 选项，123，399
AWS（Amazon Web Services）
访问键，498
附加信息，516
背景信息，487
云工具，493–496
CloudFront，499
获取账号，497
获取资格证书，498
入门，496–511
登录及密码，498
安全问题，501
SOAP 支持，498
技术概览，488–492
磁盘，511–516
实例，500
AWS 管理控制台
创建实例，500
EBS 快照，514
EBS 卷，512–513
功能，493
启动实例，508–511
登录的要求，498
运行实例，501–508

AWS 多因素认证，501

B

备份
 应用日志，426
 自动化，446–448
 克隆 master，31
 数据分片问题，181
 示例，442
 预期，422
 形成归档计划，423
 重要性，5，420–421
 Innobackup 脚本，428
 InnoDB Hot Backup 应用，425–428
 逻辑备份与物理备份，422
 LVM 支持，432–436
 方法的比较，437
 Mysqldump 工具，430–432
 联机方法，33，108
 物理文件拷贝，428–430
 PITR，439–445
 复制，438
 向外扩展，148
 快照，434
 系统可用性，6
 术语，419
 XtraBackup，432
带宽，复制监控，367，383
基线，形成，247
BEGIN 语句，75，234
Begin_load_query 事件，57
基准函数，319，340
基准，318–319
最佳实践
 云计算，520–522
 数据库优化，331–339
 高性能，558–560
 性能提高，339–341
 故障排除，401–407
双向复制，120–124，166
二进制日志，87（参见：记录语句，mysqlbinlog 工具）
 基本类和函数，25
 克隆 master，31
 克隆 slave，33–34
 内容问题，20–22，46
 崩溃的安全性，82
 创建 slave，30
 执行顺序，216
 过滤，59-61
 功能，13，17
 记录事务，75–81
 维护复制的位置，212–214
 管理，81–87
 操作系统，26
 选项和变量，98–100
 复制示例，18–20
 复制线程，371
 使用脚本进行克隆操作，35–36
 安全性，64
 Server 类，26–28
 服务器日志，315
 服务器角色，28–30
 结构问题，20–22，46–50
 故障排除，392
 --verbose 选项，564
二进制日志事件
 定义，46
 记录语句，61-66
 结构问题，46-50

故障排除，389-391
Binlog 转储线程，371
binlog 文件格式版本，48–50，84
binlog 文件轮换，83
binlog 文件
创建，88
创建时间，98
转储内容，88
功能，47
处理顺序，92
清除，85，86
远程读取，93
共享磁盘，117
Binlog 索引文件
功能，47
清除，85，87
Binlog 注入线程，554
binlog-cache-size 选项，99
binlog-cache-size 服务器变量，84
binlog-do-db 选项，59–61，163，315，369
binlog-format 选项，230，231，553
binlog-ignore-db 选项，59–61，163，315，369
binlog-in-use 标志，81，84，96
binlog-max-row-event-size 选项，231
二项式尾分布，105
Blackhole 存储引擎，160，334
引导 slave，30
Bott，Ed，283
浏览器插件，494
桶，定义，491
缓冲池
定义，353
监控，360–363
Bunce，Tim，429
业务连续性（参见信息完整性）

C

CA（权威认证机构），203，204
CACHE INDEX 命令，350
Cacti 工具，252
CALL 语句，68
中心存储复制，156
权威认证机构（CA），203，204
链型拓扑，402
CHANGE MASTER 命令
云计算，518
连接下游服务器，144
IGNORE_SERVER_IDS 选项，144，398，399
Master 心跳，383
Master 状态变量，376
MASTER_HOST 选项，202
标记 slave 位置，118
复制的故障排除，400
Slave 的故障排除，393，395
使用示例，145
CHANGE MASTER TO 命令
配置复制，199
配置 slave，32
CONNECT_RETRY 参数，215
创建 slave，30
热备份，114
MASTER_SSL 选项，204
MASTER_SSL_CAPATH 选项，204
MASTER_SSL_CERT 选项，204
MASTER_SSL_KEY 选项，204
权限问题，16
复制状态信息，207

Slave 提升，129，134
使用示例，15
通道，定义，554
校验和，251
循环复制
描述，152
功能，142-146
忽略服务器，572
拓扑，403
故障排除，398
克隆函数，35
克隆操作
主服务器，30，31
脚本，35-36
从服务器，33–34
关闭恢复映像，441
云计算
架构支持，480–483
AWS 案例学习，484，487–516
最佳实践，520–522
特点，479
定义，478
部署模型，480
成本问题，483–484
弹性，482
网格计算，481
MySQL 企业版，472
开源，522
潜在价值，485
复制，517–520
服务模型，479
软件库，483
事务计算，482
用例，484
供应商支持，486
虚拟化，481
聚簇索引，353
评论
哈希标记，90
检索，178
COMMIT 语句
记录事务，75
查询事件，234
slave 提升，132
两阶段提交，150
commit_and_sync 函数，186，190
压缩表，345，347
concat 函数，319
配置文件
最佳实践，404
二进制日志，47，59–61，165
配置 master，13–14
配置 slave，15
管理复制，24
选项支持，58·
Server 类，26
SSL 问题，204
复制的故障排除，400
CONNECTION_ID 函数
基于行的复制，229
会话特定的，216
thread ID，56，57
connect_to 帮助函数，157
一致性（参见数据一致性）
控制台应用程序，271
-- 控制台选项，315
限制，最佳实践，337
上下文事件
记录查询，51–57
SQL 线程，217–220

应急计划（高可用性）
定义，104
灾难恢复，107，417
master 故障，106
规划问题，106
中继服务器故障，107
slave 故障，106
CPU-bound 进程，249
创建基本任务向导，43
CREATE FUNCTION 语句，69
CREATE INDEX 命令，538
CREATE PROCEDURE 语句，67
CREATE ROUTINE 权限，70
CREATE 语句
Binlog 事件，48
ENGINE 参数，334
记录事务，76
CREATE TABLE IF NOT EXISTS 语句，46，135
CREATE TABLE 语句
隐式提交，135
记录日志，51
CREATE TRIGGER 语句
DEFINER 从句，65
安全问题，63
CREATE USER 权限，16
Create_file_log_event，58
资格证书
获取，498
实例，500
cron 工具，253，268
crontab 文件，42
CSV 存储引擎，334
CURDATE 函数，53
当前数据库
二进制日志过滤器，59–61，567
记录查询，51，52
查询事件，97
USE 语句，60
当前时间，记录查询，52，53
CURRENT_ID 函数，52
CURRENT_USER 函数，229，231
CURTIME 函数，53
周期函数，228
Cygwin 环境，429

D

仪表盘，453
（参见 Enterprise Dashboard）
定义，453
MEM 支持，456
Data Collector Sets，287
数据一致性
异步复制，150，151
级联部署，187–193
管理，185–187
MyISAM 问题，82
非级联部署，185–187
数据定义语言（DDL）语句
备份，430
记录语句，51
记录事务，76
数据管理器，481
数据操纵语言（DML），50
数据保护，410
数据恢复，439
（参见灾难恢复）
备份和恢复，420–424
示例，440
重要性，419

PITR，439–445
复制，438
术语，419
数据分片
常见用法，168
数据库问题，175
动态，171，179–180
示例，173–184
负载均衡，168，171–173，180–184
管理，168
分片之间的移动，173
移到不同节点，171–173
命名问题，170
分区键 / 函数，175–177
分区数据，170
过程概览，166–169
读取分片，177–179
分片的表示，169
Slave 滞后，385
静态，170
写数据，149
数据挖掘查询，37
数据库
拷贝文件，108
性能评估，320–331
优化，331–339，368
性能信息，320
崩溃后复制，222–225
分片，175
--datadir 启动选项，344，435
Date，C. J.，320
dd 命令，521
DDL（数据定义语言）语句
备份，430
记录语句，51
记录事务，76
对表进行碎片整理，348
延迟的 slave，5
DELETE 语句
LIMIT 从句，229，240
记录日志，50
非事务性变化，225
存储过程，66
WHERE 从句，46
Delete_rows 事件，232，237
部署
定义，24
级联，187–193
非级联，185–187
DESCRIBE 命令，320
df 命令，264
菱形配置，226
差异备份，422
灾难避免
热备份，11，111–114，148
远程复制，148
灾难恢复，439
（参见数据恢复）
应急计划，107
目标，414
高可用性，418，550–551
信息完整性，412，413–419
规划问题，415
实践，417
slave 服务器，222–226
工具和策略，417–419
工作流问题，415–417
磁盘使用情况
AWS 问题，511–516
Linux/Unix 环境，261

Mac OS X 环境，274
监控，247，250
优化，344
磁盘使用分析器，264
disk-bound 进程，250
分布式数据，冗余，531
分布式复制块设备（DRBD），118
分布式事务处理，79–81
分布成本，管理复制，153
DML（数据管理语言），50
DRBD（分布式复制块设备），118
DROP 命令，335
DROP INDEX 命令，538
DROP TABLE IF EXISTS 语句，46
DROP TABLE 语句，22，135
双主设置
active-active 设置，115，116，120–124
active-passive 设置，115，116，119
双向复制，120–124，166
定义，6
描述，152
高可用性，115–124
管理复制，23
复制磁盘，118
共享磁盘，117
动态分片，171，179–180

E

ec2-add-key-pair 命令，495
ec2-attach-volume 命令，515
ec2-authorize 命令，510
ec2-create-snapshot 命令，516
ec2-create-volume 命令，515
ec2-delete-snapshot 命令，516
ec2-delete-volume 命令，516
ec2-describe-images 命令，496，509
ec2-describe-snapshots 命令，516
ec2-describe-volumes 命令，515
ec2-detach-volume 命令，515
ec2-run-instances 命令，496，509
ec2-start-instances 命令，496
ec2-stop-instances 命令，496
ec2-terminate-instances 命令，496，510
弹性负载均衡，489
Elastic MapReduce，488
Elasticfox（Firefox 插件），494
弹性，云计算，482
EmptyRowError 异常，25
_enable_binlog 帮助方法，136
加密
安全问题，64
SSL 支持，202
Enki 咨询，486
企业仪表盘
Advisors 标签，468
综合服务器图，466
功能，456
热图，464
查询分析器，459
Replication 标签，468
服务器细节，466
epoch，定义，531
Error 类，25
错误处理
日志消息，282
记录语句，72–75
错误日志，314
事件调度器，42
事件查看器，281–283

事件，46
（参见二进制日志事件）
影响复制，81
定义，71
DEFINER 从句，71
执行，236–237
过滤，163–164，221–222
处理基于行的复制，232–236
事故，85
解释，94–98
记录语句，61–66，71
对象定义，180
分区到 slave 上，165
口令问题，64
跳过，217，221–222
slave 处理，215–222
SQL 线程处理，217
触发器，238–239
过度滞后，396
Execute_load_query 事件，57
Execute_log_ 事件，58
expire-logs-days 选项，86，98
EXPLAIN 命令
最佳实践，339
执行查询，458
功能，320–327
索引，331
使用示例，313
EXTENDED 关键字，323
extended-status 命令，301
外部复制，552

F

Faroult，Stephane，320
fdisk 命令，264
Federated 存储引擎，335
fetch_global_trans_id 函数，140
fetch_master_pos 函数，35，186，191
fetch_relay_chain 函数，191
fetch_remote_binlog 函数，40，138
fetch_slave_pos 函数，35
fetch_trans_id 函数，190
文件 ID，57
文件系统
协调同步，83
磁盘使用情况，250
记录变化，51，76
逻辑卷，434
内存问题，250
对象定义，180
读取远程文件，93
快照支持，108
过滤
当前数据库，567
将事件分配到 slave，165
复制事件，163–164
基于行的复制，240
向外扩展，149
跳过事件，221–222
SQL 线程，217
过滤，二进制日志过滤器，59–61
Final 角色，29
find_datetime_position 函数，40
FLUSH LOGS 命令
binlog 文件支持，47
功能，18，84
监控 master 服务器，374
权限问题，16
二进制日志故障排除，392
FLUSH QUERY CACHE 命令，300

FLUSH STATUS 命令，307
FLUSH TABLES 命令，429
FLUSH TABLES WITH READ LOCK 命令
 克隆 master，31
 EBS 快照，514
 InnoDB 注意事项，34
 LVM 支持，436
 暂停复制，406
 释放锁，27
外键，338
格式描述事件
 Binlog 事件结构，47，48–50
 功能，19，20
 header 限制，97
 I/O 线程，216
 二进制日志，20，84
 解释，97
 打印，89
 XA，81
碎片，定义，552
free 命令，253，259
fsync 调用，223，224

G

通用查询日志，314
getArticlesForUser 函数，177，182
getCommentsForArticle 函数，178
getServerConnection 函数（PHP），157
全局冗余，530，557
全局事务 ID
 环形复制，144，145
 数据一致性示例，188，191
 定义，130
 slave 提升，130–135，137，139
GNOME 桌面项目，264
Gnome 系统监控器，252
Governor，James，478
GRANT OPTION 权限，16
图形用户接口（GUI），268，302
grep 命令，258，509
网格计算，481
组，定义，48
group_by_event 函数，138
group_by_trans 函数，138
GUI（图形用户接口），268，302

H

硬件
 数据丢失，421
 节点恢复，551
井号（#），90
HA_ERR_KEY_NOT_FOUND 错误，399
健康保险携带和责任法案（HIPAA），412
心跳，383，551，571
热图，464
海森堡不确定性原理，306
Hibernate 分片（Google），168
级联部署，187–193
级联复制
 定义，159
 设置中继服务器，160
高可用性
 获取，547–550
 备份，6
 应急计划，104，106–107
 灾难恢复，418
 信息完整性，412
 MyISAM 存储引擎，352

节点恢复，551
过程，104，107–146
冗余，103，104
复制，6，552–556
系统恢复，550–551
HIPAA（健康保险携带和责任法案），412
HiveDB 分片实现，168
水平分割（参见数据分片）
host-bin 选项，99
hostname-bin 选项，14
热备份
定义，11
灾难避免，11，148
高可用性，111–114
混合拓扑，403

I

IA（信息保障）
附加信息，410
定义，410
重要性，411
相关实践，410
IaaS（基础设施即服务）
定义，479
虚拟化，481
ibbackup 工具
--apply-log 选项，426
核心业务，425
执行备份，425
恢复数据，427
--uncompress 选项，426
IBM Blue Cloud，486
ifconfig 命令，265
IGNORE LEAVES 从句，350
映像，定义，481
Incident 事件类型，86
incident 事件，定义，85
增量备份，422
索引文件（参见 binlog 索引文件）
索引
最佳实践，331，338
集群，353
精细调优，320
预装载，349
查询，294，299
信息保障（参见 IA）
信息完整性
备份和恢复，420–424
数据恢复，419
定义，410，411–412
灾难恢复，412，413–419
高可用性，412
概述，411
信息重要性，410
INFORMATION_SCHEMA 数据库，363–364
基础设施即服务（参见 IaaS）
--initial 选项，543
--initial-start 选项，543
init_file 选项，389
innobackup 脚本
功能，427
选项支持，427
执行备份，428
恢复数据，428
InnoDB Hot Backup
应用，425–428，437
克隆 master，31
InnoDB 存储引擎

架构特点，353
数据分片，175
双主设置，118
功能，334
处理行锁，182
Hot Backup，33，108，425–428
提升性能，352
INFORMATION_SCHEMA 数据库，363–364
InnoTop 系统活动报告，317
监控机制，357–359
监控缓冲池，360–363
监控日志文件，359
监控表空间，363
OPTIMIZE TABLE 命令，330
参数支持，365
恢复问题，119
半同步复制，124
SHOW ENGINE INNODB STATUS 命令，354–356
Slave 提升，131
快照支持，108
XA 支持，83
innodb_fast_shutdown 选项，365
innodb_thread_consistency 选项，365
InnoTop 系统活动报告，317
INSERT INTO 语句，135
INSERT 语句
调用触发器，65
LIMIT 从句，229，240
记录日志，50
非事务性改变，73，225
存储函数，69
存储过程，66
使用示例，90，219
INSERT_ID 会话变量，218
实例
定义，481
启动，508–511
运行，501–508
存储问题，511
实例，资格证书，500
整型数据，解释，94
内部复制，552
Internet，运行复制，202–206
Intvar 事件
功能，54，218
mysqlbinlog 支持，90，91
存储过程，68
库存评估，416
I/O 线程
功能，200
处理断开的连接，215
管家，216
复制，371
启动和停止，201
状态问题，210–212
同步，224
I/O 传输率，262
I/O-bound 进程，251
I/O-starved 进程，251
ionice 命令，255
iostat 命令，253，255，261
itertools 模块，228

J

Joyent（供应商），487

K

KDE System Guard，252

内核，内存，250
key cache
 创建，350
 监控，348
 多个 key cache，350
 预装载，349
KILL 命令，295，301

L

LAST_INSERT_ID 函数
 全局事务 ID，132
 记录查询，52，53，54
LAST_INSERT_ID 会话变量，218
Layered Technologies（供应商），487
法律要求，数据保护，7
L' Hermite，Pascal，320
LIKE 从句，295
LIMIT 从句，229，240
Linux 类，26
Linux 环境
 自动监控，268
 磁盘使用情况，261–264
 一般系统统计数据，266
 InnoDB Hot Backup 应用，425
 LVM 支持，31，34，108
 管理复制，24
 内存使用情况，259–261
 监控，246，253–268
 网络活动，265
 进程活动，253–258
Linux High Availability 项目，118
负载均衡
 应用程序级，155–159
 数据分片，168，171–173，180–184
 弹性，489
 管理复制，152
 进行读取，148
 Slave 滞后，385
 进行写，148
LOAD DATA INFILE 语句
 数据分片示例，184
 处理当前数据库，52
 LOAD_FILE 函数，72
 记录语句，57–58
 记录事务，76
LOAD INDEX 命令，349
LOAD_FILE 函数，72，231
Load_log_event，58
LOCAL 关键字，329，330
本地查询处理器，537
局部冗余，530，557
LOCK TABLES 命令，172
锁
 减少争用，385
 释放，27
 行，183
LOCK_log 互斥锁，50
--log 启动选项，314
log-bin 选项
 控制 binlog 文件，47
 功能，13，99，315
 Server 类，26
 slave 提升，129
log-bin-index 选项
 控制 binlog 文件，47
 功能，13，99，315
 Server 类，26
log-bin-trust-function-creators 选项，71，99
--log-error 启动选项，315

--log-output 启动选项，314
log-slave-updates 选项
双向复制，121
二进制日志，393
级联复制，160
热备份，112
slave 提升，129，130，136
--log-slow-queries 启动选项，314
--log-slow-slave-statements 选项，314
日志文件
应用到备份，426
最佳实践，404
控制台应用程序，271
实践查看器支持，281
InnoDB 存储引擎，353
监控，359
RESET_SLAVE 命令，199
服务器日志，313
复制的故障排除，400
数据表的故障排除，391
记录语句
二进制日志过滤器，59–61
DDL 语句，51
DML 语句，50
异常处理，72–75
事件，61–66，71
LOAD DATA INFILE 语句，57–58
LOCK_log 互斥锁，50
非事务性改变，72–75
查询事件，51–57
特殊结构，71
存储函数，61，66，69–70
存储过程，61，66–68
存储程序，61–66
存储例程，61–66
触发器，61–66
记录事务
隐式提交，76
启动事务，75
事务缓存，76–79
XA 支持，79–81
逻辑备份，422
逻辑卷管理器（参见 LVM）
逻辑卷，434
Loukides，Mike，262
ls 命令，264
Lua 编程语言，154
lvcreate 命令，434
LVM（逻辑卷管理器）
备份方法比较，437
克隆 slave，34
克隆 master，31
功能，432
快照支持，108
使用问题，433–436
lvremove 命令，435
lvscan 命令，435

M

Mac OS X 环境
Activity Monitor，273–276
控制台应用程序，271
监控，246，268–276
System Profiler，268–271
Machine 类，26
管理层介入，416
管理二进制日志，81–87
Martelli，Alex，140
master 转储线程，200
master 过滤器

创建，163
定义，163
master 心跳，383
master 日志信息文件
刷新，224
功能，198
操作 slave 线程，201
复制状态信息，212
Master 角色
定义，29
replicate_from 函数，35
master 服务器
检查状态，403
环形复制，142–146
克隆，30，31
配置，13
连接到 slave，14，15
创建，7
创建复制用户，14
延迟的 slave，5
双主设置，6，23，115–124
故障处理，106，109
级联复制，159
监控，372–376
监控线程状态，371
多主问题，399
复制概述，5
克隆操作脚本，35
server 角色，28–30
状态变量，376
切换，109，112–114
技巧和窍门，568–570
故障排除，388–393
两阶段提交，150
升级，110
master-connect-retry 选项，215
master-retry-count 选项，215
MASTER_POS_WAIT 函数
数据一致性示例，185，187
功能，40，172
中继日志处理，213
max-allowed-packet 选项，58，397
max-binlog-cache-size 选项，100
max-binlog-size 选项，100
Maxia，Giuseppe，155
MD5 函数，220
MEM（参见 MySQL Enterprise Monitor）
memcached 技术，156，171
内存
调整时的注意事项，250
Mac OS X 环境，274
监控，247，249，259–261
节点恢复，551
故障排除，389，395
Memory 存储引擎，335
memory-bound 进程，249
Merge 存储引擎，335
MERGE 视图，122
Microsoft Azure，483
微软管理控制台的控制单元，42
任务声明，416
混合节点复制，231
Mollinaro，Anthony，320
监控，292
（参见性能问题）
自动化，268
好处，247
缓冲池，360–363
类别 246
定义，246

磁盘使用情况，247，250，261–264，274
一般系统统计数据，266
InnoDB 存储引擎，352–365
key cache，348
Linux 环境，246，253–268
日志文件，359
Mac OS X 环境，246，268–276
master 服务器，372–376
内存，247，249，259–261，274
MyISAM 存储引擎，344–352
MySQL Administrator，381
MySQL 企业版，463–469
MySQL 服务器，292–319
网络活动，248，251，265，275
作为预防性维护，288
进程活动，253–258
处理器，247，248
复制，367–386
半同步复制，127
slave 滞后，383
slave 服务器，376
表空间，363
工具，252
Unix 环境，246，253–268
使用示例，7
Windows 环境，246，276–288
监控代理，457，462
MONyog 工具，317
mount 命令，435
mpstat 命令，253，255，257
多通道复制，554，567
多主拓扑，399，403
多源复制，226–228，566
Musumeci，Gian-Paolo D.，262
互斥，356
myisam ftdump 工具，345
MyISAM 存储引擎
压缩表，347
一致性问题，82
对表进行碎片整理，348
双主设置，118
功能，334
处理行锁，183
高可用性，352
提升性能，344
监控 key cache，348
非事务性改变，73，75，225
OPTIMIZE TABLE 命令，330
优化磁盘存储，344
参数支持，351
预装载 key cache，349
Query Cache，298，307
恢复问题，119
slave 提升，131
按索引排序的数据表，347
数据表的故障排除，397
调整数据表，345–346
myisam-recover 选项，392
myisamchk 工具
对表进行碎片整理，348
功能，345–346
按索引排序的数据表，347
myisamlog 工具，345
myisampack 工具，345，347
MySAR 系统活动报告，316
MySQL
附加信息，8
版本问题，24
MySQL Administrator

Connection Health 标签，303
功能，302
Key Efficiency 图，307
Memory Health 标签，306
页面工具，311
Query Cache Hitrate 图，307
复制的监控，381
Server Variables 标签，309
Status Variables 标签，310
Traffic 图，304
MySQL 集群
架构基础，532–538，554
支持提交，151
数据节点，543
数据存储，533–536
示例配置，539–547
特点，528–529
功能，526
入门，539–541
高可用性，547–556
高性能，557–560
日志处理，531
管理节点，541
NDB 管理控制台，542
联机操作，537
分区，536
冗余，530，531，557
reload 事件，86
复制，566
复制，553
关闭集群，546
SQL 节点，544
启动，541–546
术语和组件，526
测试集群，546
事务管理，537
典型配置，527
mysql 数据库
记录事务，76
对象定义，180
MySQL 企业版
警告细节信息，464
背景信息，452
云计算，472
组件，456–460
安装监控代理，462
安装，454–455，460–462
监控，463–469
产品支持，459
Query Analyzer，469–472
订阅级别，453
使用问题，460
MySQL Enterprise Backup，425
MySQL Enterprise Monitor
附加信息，252
顾问，457
背景信息，453
Enterprise Dashboard，456
功能，452，456
安装，455
监控代理，457，463
Query Analyzer，458
MySQL Enterprise Server，456
MySQL Forge，155
MySQL Migration Toolkit，302
MySQL 监控，294
MySQL Monitor and Advisor（MONyog）
工具，317
MySQL Proxy
数据分片，169

负载均衡，154
多主复制，565
报告统计数据，469
MySQL Python
添加中继服务器，161
附加信息，8
常见复制任务，36–43
报告处理，40
切换处理，114
管理复制，23–25
PITR，443–445
slave 提升，135–141
MySQL Query Browser，312–313
MySQL 服务器
基准套件，318–319
通信性能，293
GUI 工具，302
MySQL Administrator，302–312
MySQL Query Browser，312–313
mysqladmin 工具，300–302
性能监控，293
服务器日志，313
SQL 命令，294–300
第三方工具，316–318
MySQL System Tray Monitor，313
mysql 工具，32
mysql.com 停机事件，110
mysqladmin 工具
命令支持，300
--relative 选项，301
--sleep 选项，301
mysqlbinlog 工具
--base64-output=never 选项，89
基本用法，88–93
--force 选项，38
--force-if-open 选项，89
功能，87，297
--hexdump 选项，94，95
解释评论，90
解释事件，94–98
PITR，439
pseudo_thread_id 变量，57
--read-from-remote-server 选项，93
读取远程文件，93
--short-form 选项，89，90
--start-datetime 选项，38，92
--start-position 选项，92
--stop-datetime 选项，38，93
--stop-position 选项，92
复制的故障排除，400
使用示例，39
查看错误代码，75
支持通配符，92
mysqldump 工具
备份方法的比较，437
克隆 slave，34
克隆 master，31
功能，430–432
选项支持，431
快照，108
mytop 工具，316

N

Nagios 工具，252，288
NAME_CONST 函数，68
美国国家标准技术研究院（NIST），479
NDB（网络数据库），526
NDB 管理控制台，537，542，547
--NDB-connectstring 选项，542，544
--NDB-nodeid 选项，543，544

NDBcluster 选项，544
NDB_binlog_index 表，554
NDB_restore 工具，538
netstat 命令，253，265
网络活动
 Linux/UNIX 环境，265
 Mac OS X 环境，275
 监控，248，251
网络数据库（NDB），526
network-bound 进程，251
nice 命令，255
NIST（美国国家标准技术研究院），479
节点恢复，551
非级联部署，185–187
非事务性改变
 避免问题，79
 错误处理，72–75
 隐式提交，76
 记录日志，77–79
 保护，225
 基于行的复制，229
 故障排除，392，397
NoOptionError 异常，25
规范化，331，338
NOT NULL 限制，338
NotMasterError 异常，25
NotSlaveError 异常，25
NOW 函数，52，53
NO_WRITE_TO_BINLOG 关键字，329，330

O

on_gid 函数，139
打开恢复映像，441
开源云计算，522
操作系统，252
 （参见特殊系统）
 类方法，26
 管理复制，24
 监控方案，252
 节点恢复，551
OPTIMIZE TABLE 命令
 最佳实践，339
 对表进行碎片整理，348
 功能，330，345
oracle 算法，574
ORDER BY 从句，241，347
总传输率，250

P

PaaS（平台即服务），480
页面缓存，83
分页技术，249
分区函数
 常用模式，170
 数据库分片，175–177
分区键
 创建，170
 数据库分片，175–177
分区，167
 （参见数据分片）
 数据分片，170，175–177
 定义，433
 发给 slave 的事件，165
 MySQL 集群，536
口令
 AWS 需求，498
 Master 日志信息文件，199
 安全问题，63，64
爱国者法案，412

暂停复制，406
PBXT 事务引擎，119
峰值负载，处理，153
每个进程的传输率，250，251
Percona 开源提供者，432
性能问题，248
（参见监控）
最佳实践，339–341，558–560
数据挖掘，37
数据库，319
数据库对象操作，51
定义，292
高性能，557
InnoDB 存储引擎，352
MyISAM 存储引擎，344
MySQL 集群，557–560
MySQL 服务器，292–319
优化视图，122
复制，341，367
生成报告，12，148
同步复制，151
调整数据表，345–346
Performance Monitor，285–288
Perl 语言，318，427
PHP 编程语言，156
物理备份，422
物理文件拷贝，428–430，437
物理卷，433
pid-file 选项
功能，13，99
Server 类，26
PITR（即时恢复）
复制备份，439
备份过程，442
二进制日志，17，51，165，315
定义，4
过滤问题，163
FLUSH LOGS 命令，84
InnoDB Hot Backup，428
Python，443–445
恢复示例，440
恢复映像，441
复制，369
复制错误后恢复，439
平台即服务（PaaS），480
pmap 命令，253，259
即时恢复（参见 PITR）
轮询，189
pool_add 函数，158
pool_del 函数，158
pool_set 函数，158
Position 类，25
提交头，48–50，98
主键，338
主服务器，116，117
私有键，203
权限
配置复制，14，16
读取远程文件，93
安全性和二进制日志，64
设置线程 ID，57
存储函数，70
主动监控，247
过程（高可用性）
最佳实践，406
环形复制，142–146
考虑的问题，108–110
定义，104
双主设置，6，23，115–124
热备份，11，111–114

半同步复制，116，124–127
Slave 提升，109，127–141
进程 ID
识别，258
临时表，56
TLS 支持，220
进程
分配权限，249，255
CPU-bound，249
定义，248
disk-bound，250
I/O-bound，251
I/O-starved，251
杀死失控的进程，248
memory-bound，249
监控活动，253–258
network-bound，251
processor-bound，249
删除不必要的进程，248
重新调度，249
解决超载，248
processlist 命令，301
处理器，监控，247，248
processor-bound 进程，249
Promotable 类，136
promote_slave 函数，140
代理
定义，153
分布式查询，154
ps 命令，253，257
pseudothread ID，221
pseudo_thread_id 服务器变量，57
公共证书，203
PURGE BINARY LOGS 命令，47，86，400
清除索引文件，85
pvcreate 命令，434
pvscan 命令，434
Python（参见 MySQL Python）

Q

查询
分析，153
最佳实践，559
数据分片，171
数据挖掘，37
分布式，153，154
EXPLAIN 命令，458
提升性能，340
手动执行，405
slave 滞后，385
故障排除，391，392，394
Query Analyzer
功能，458，469–472
故障排除，463
query cache
最佳实践，336，560
功能，298
MySQL Administrator，306
服务器变量，299
查询事件
binlog 事件结构，48
上下文事件，217
当前数据库，97
执行上下文，51，53–54
功能，19
解释，94–97
记录日志，51–57
mysqlbinlog 示例，91
读取远程文件，93

基于行的复制，234
线程 ID，56，221

R

Rackspace（供应商），487
RAID（廉价磁盘冗余阵列），412
Rand 事件，54，218
RAND 函数
上下文事件，53
功能，52
Rand 事件，54，218
反应监控，247
read-only 选项，100
读取数据
避免陈旧数据，189
数据分片，177–179
负载均衡，148
远程文件，93
向外扩展，149
线程局部对象，220
恢复映像，441，444
恢复点目标（RPO），419，423
恢复时间目标（RTO），419，424
冗余（高可用性）
定义，103
MySQL 集群，530，531，557
原理概述，104
廉价磁盘冗余阵列（RAID），412
Reese，George，478
中继日志
配置 slave，15
事件的执行，236
维护复制的位置，212–214
结构，196–200
故障排除，397
中继日志信息文件
功能，198，199
操作 slave 线程，201
复制状态信息，213
线程同步，224
中继服务器
加入 Python，161
故障处理，107
级联复制，159
设置，160
同步，188
relay-log-index 选项，15
Reliability Monitor，283
RELOAD 权限，16
renice 命令，255
REORGANIZE PARTITION 命令，538
REPAIR TABLE 命令，345
Replicant 库
故障处理，118
负载均衡函数，158
多源复制，228
分片的重新均衡，181
副本，定义，531
replicate-do-db 选项
数据分片，169，172
复制的监控，369
slave 过滤器，164
线程处理，217
replicate-do-table 选项
复制的监控，370
slave 过滤器，164
线程处理，217
replicate-ignore-db 选项
复制的监控，369
slave 过滤器，164

线程处理，217
replicate-ignore-table 选项
复制的监控，370
slave 过滤器，164
线程处理，217
replicate-rewrite-db 选项，370
replicate-same-server-id 选项，121，370
replicate-wild-do-table 选项
复制的监控，370
线程处理，217
replicate-wild-ignore-table 选项
复制的监控，370
Slave 过滤器，164
线程处理，217
replicate_from 函数，35
复制，103
（参见高可用性；基于行的复制；向外扩展；基于语句的复制）
架构基础，196–202
异步，6，150–152
备份和恢复，438
基本步骤，12–16
双向，120–124，166
二进制日志示例，18–20
业务功能，5
环　形，142–146，152，398，403，572
常见用法，148
配置权限，14，16
定义，5
EC2，517–520
Enterprise Dashboard，468
过滤事件，163–164
处理断开的连接，214
级联的，159–161
高可用性，6，552–556
提升性能，341，367
包容性和排他性，368–370
管理拓扑，152–159
使用 Python 管理，23–25
混合模式，231
监控 master 服务器，372–376
监控 slave 服务器，376
多道，554，567
多源，226–228，566
MySQL Administrator，381
MySQL 集群，553
MySQL Proxy，565
暂停，406
执行常规任务，36–43
PITR，439–445
过程概述，17
再植表，564
报告 bug，407
通过 Internet 运行，202–206
支持脚本的，573
分段，570
半同步，116，124–127
服务器设置，368
slave 安全性和恢复，222–226
slave 处理事件，215–222
状态信息，206–214
同步，150，151
时间延迟，572
技巧和窍门，563–574
故障排除，393，398，399，406
REPLICATION CLIENT 权限，16
REPLICATION SLAVE 权限
读取远程文件，93
使用建议，14，16，64

复制线程，200，371–372
复制拓扑（参见拓扑）
再植表，564
生成报告
 性能问题，12，148
 过程概述，37–43
 复制 bug，407
 向外扩展，148
report-host 选项，207，210，381
report-password 选项，207
report-port 选项，207
report-user 选项，207
REQUIRE SSL 选项，395
RESET MASTER 命令
 Binlog 文件支持，47
 功能，22
 slave 提升，129
 使用示例，22
RESET SLAVE 命令
 均衡分片，172
 Master 日志信息文件，199
 STOP SLAVE 命令，22
 使用示例，22
资源管理器，79
响应时间，监控，247
重启，最佳实践，405
恢复进程
 错误复制后，439
 期望，422
 形成归档计划，423
 ibbackup 工具，427
 innobackup 脚本，428
 LVM 支持，436
返回值，存储例程，69
环形拓扑，403
风险评估，416
Role 类
 create_repl_user 方法，29
 disable_binlog 方法，29
 enable_binlog 方法，29
 功能，28
 imbue 方法，29
 set_server_id 方法，29
 unimbue 方法，29
ROLLBACK 语句，75
Romanenko，Igor，430
rotate 事件
 二进制日志，20
 Binlog 事件结构，47，49–50
 binlog-in-use 标志，84
 功能，19，21
 header 限制，97
 I/O 线程，216
轮换 DNS，156
轮换多源复制，226
Row 类，27
行事件，235，236
基于行的复制
 配置选项，230
 定义，17
 事件执行，236–237
 事件和触发器，238–239
 事件处理，232–236
 过滤，240
 功能，229–230
 记录语句，50
 混合模式复制，231
 非事务性改变，72
 基于语句的复制，229
 技巧和窍门，565

rpl-semi-sync-master-enabled 选项，126
rpl-semi-sync-master-timeout 选项，126
rpl-semi-sync-master-wait-no-slave 选项，126
rpl-semi-sync-slave-enabled 选项，126
rpl_semi_sync_master_clients 选项，127
rpl_semi_sync_master_status 选项，127
rpl_semi_sync_slave_status 选项，127
RPO（恢复点目标），419，423
RTO（恢复时间目标），419，424

S

SaaS（软件即服务），480
Salesforce.com，487
SAN（存储区域网络），117，491
sar 命令，253，255，262–263
法案（SOX），412
保存点，334
向外扩展
 异步复制，150–152
 常见用法，148
 数据一致性，185–193
 数据分片，166–184
 定义，6，147
 级联复制，159–161
 管理复制拓扑，152–159
 读取数据，149
 专用 slave，162–165
 写数据，149
向上扩展，定义，147
scan_logfile 函数，139
任务调度
 UNIX 上，42
 Windows Vista 上，42
Schlossnagle，Theo，8
Schwartz，Baron，8，148，558
脚本
 克隆操作，35–36
 复制，573
SCSI，118
检索
 控制台应用程序，271
 基于行，237
辅助服务器，116
安全套接层（参见 SSL）
安全性
 AWS 支持，501
 二进制日志，64
 IA，410
 日志文件消息，281
 监控问题，246
 口令问题，63，64
 复制线程，70
SELECT MASTER_POS_WAIT 函数，406
SELECT 语句
 数据一致性示例，191
 EXPLAIN 命令，320
 LIKE 从句，295
 LIMIT 修饰符，157
 数据均衡示例，157
 日志问题，46
 非事务性改变，74
 ORDER BY RAND() 修饰符，157
 半同步复制，127
 存储函数，69
 内存表的故障排除，395
 查询的故障排除，394
 WHERE 从句，336
半同步复制

配置，125–127
功能，116，124
监控，127
序列化事务的执行，46
Server 类
connect 方法，27
disconnect 方法，27
fetch_config 方法，28
重要参数，26–28
replace_config 方法，28
克隆操作的脚本，35
sql 方法，27
ssh 方法，27
start 方法，28
stop 方法，28
服务器 ID
环形复制，144
配置 master，13
双主设置，118，121
Role 类，29
Server 类，27
slave 提升，131
server 角色
创建，161
功能，28–30
服务器版本，84
server-id 选项
连接超时，394
功能，14
SET GLOBAL 命令，314
SET 语句
创建 key cache，350
使用示例，64
分片 ID，168
分片技术（参见数据分片）
shardNumber 函数，177
shell 命令
管理复制，24
Server 类，27
SHOW BINARY LOGS 命令
功能，38，297
监控 master，373
监控 slave，379
SHOW BINLOG EVENTS 命令
上下文事件，54
错误代码，75
功能，297
级联复制，159
监控 master 服务器，374–376
监控 slave 服务器，379
复制的故障排除，400
使用实例，18，21
SHOW COLUMNS FROM 命令，320
SHOW ENGINE INNODB MUTEX 命令，356
SHOW ENGINE INNODB STATUS 命令
功能，354–356
InnoDB 监视器，357
监控缓冲池，360
监控表空间，363
SHOW ENGINE LOGS 命令，296
SHOW ENGINE STATUS 命令，296
SHOW ENGINES 命令，296，332
SHOW FULL PROCESSLIST 命令，316
SHOW GRANTS FOR 命令，400
SHOW INDEX 命令，328
SHOW INDEX FROM 命令，294
SHOW MASTER LOGS 命令
克隆 master，31
全局事务 ID，134

复制状态信息，208
SHOW MASTER STATUS 命令，133
SHOW MASTER STATUS 命令
备份过程，442
最佳实践，403
克隆 master，31
数据一致性示例，185，187
功能，297
Master 状态变量，376
权限问题，16
复制状态信息，208
报告 bug，407
SHOW MASTER LOGS 命令，133
复制的故障排除，399，406
使用示例，22，114，373
SHOW PLUGINS 命令，294
SHOW PROCESSLIST 命令
功能，214，294
监控 slave 滞后，384
监控线程，371，372
mytop 工具，316
复制的故障排除，400
SHOW RELAYLOG EVENTS 命令，297，380
SHOW SLAVE HOSTS 命令
功能，297
Slave 状态变量，381
状态信息，206
复制的故障排除，400
SHOW SLAVE STATUS 命令
最佳实践，403
环形复制示例，145
克隆 slave，34
云计算，518
数据一致性示例，192
功能，297
监控滞后，384
监控 slave，377–379
权限问题，16
复制状态信息，209，212
报告 bug，407
Slave 状态变量，381
复制的故障排除，399，406
Slave 的故障排除，393，396
SHOW STATUS 命令
控制 key cache，348
功能，295
限制输出，295
MySAR 系统活动报告，316
MySQL Administrator，311
mytop 工具，316
读取变量，127
SHOW TABLE STATUS 命令，295
SHOW VARIABLES 命令
控制 key cache，348
功能，295，314
限制输出，295
MySAR 系统活动报告，316
SHOW WARNINGS 命令，323
show-slave-auth-info 选项，207
SHUTDOWN 命令，547
关闭，最佳实践，404
slave 过滤器
定义，163
过滤规则，164
slave 提升
考虑的问题，109
高可用性，127–141
Python，135–141

修正方法，129–135
传统方法，128
slave 服务器
检查状态，403
克隆，33–34
配置，15，32
连接 master，14，15
创建，7，30，108
处理滞后，384
数据库崩溃，222–225
延迟的 slave，5
事件，71
过滤复制事件，163–164
处理故障，106，109
级联复制，159
管理滞后，383
监控，376
监控线程状态，372
分区事件，165
处理事件，215–222
复制概述，5
安全性和恢复，222–226
向外扩展，162–165
克隆操作脚本，35
server 角色，28–30
状态变量，380
同步，128，150，222–225
技巧和窍门，568–570
事务，222–225
故障排除，393–398，563
两阶段提交，150
升级，109
slave 线程，200，201
（参见 I/O 线程；SQL 线程）
slave-net-timeout 选项，215
SlaveNotRunningError 异常，25
慢查询日志，314
快照
定义，433
EBS 支持，514
逻辑卷，434
方法，108
SOAP 协议，498
软件即服务（SaaS），480
软件库，483
Solaris 类，26
Solaris ZFS，108，437
SOX（萨班斯 - 奥克斯利法案），412
分裂（参见数据分片）
裂脑综合症
定义，117
DRBD，120
MySQL 集群，531
共享磁盘方案，118
SQL 线程
检查状态，214
上下文事件，217–220
过滤和跳过事件，221–222
功能，200
过程概述，215，217–222
复制，371
启动和停止，201
状态问题，210–212
同步，224
线程特定的事件，220
SQL_SLAVE_SKIP_COUNTER 变量，
221，391，392
SSH 键对，500，505
ssh 隧道模式，203
SSL（安全套接层）

Master 日志信息文件，199
监控复制，383
MySQL 支持，64
通过 Internet 进行复制，202
复制支持，204
Slave 的故障排除，395
ssl-capath 选项，204，395
ssl-cert 选项，204，395
ssl-key 选项，204，395
星型拓扑，402
START SLAVE 命令
连接 master 和 slave，15
管理 slave 线程，202
提升 slave，134
中继日志信息文件，200
Slave 状态变量，381
复制的故障排除，400
START SLAVE IO_THREAD 命令，202
START SLAVE SQL_THREAD 命令，202
START SLAVE UNTIL 命令，40，113，172
START TRANSACTION 命令，75
start_trans 函数，186，190
基于语句的复制
定义，17
过滤，240
记录语句，50
记录事务，77
语句的部分执行，240
基于行的复制，229
特殊结构，71
技巧和窍门，564
语句（参见记录语句）
静态分片，170
状态命令，301
状态变量
示例，97
海森堡不确定性原理，306
监控缓冲池，361
监控日志文件，359
监控 master 服务器，376
监控 slave 服务器，380
半同步复制，127
Stop 事件，85，216
STOP SLAVE 命令
操作 slave 线程，202
RESET SLAVE 命令，22
slave 提升，129
slave 状态变量，381
复制的故障排除，400
使用示例，22，38
STOP SLAVE IO_THREAD 命令，202，227
STOP SLAVE SQL_THREAD 命令，202，227
STOP SLAVE UNTIL 命令，40
存储区域网络（SAN），117，491
存储引擎，343
（参见特殊存储引擎）
默认，566
监控，343
概述，332–336
存储函数
定义，61，66，69
DEFINER 从句，69
INSERT 语句，69
记录语句，69–70
权限，70
SELECT 语句，69

确定特性，69
SQL SECURITY DEFINER 特性，71
SQL SECURITY INVOKER 特性，70
存储过程
提交事务，141
定义，61，66
DEFINER 从句，67，97
记录语句，66–68
存储程序
定义，61
处理事件，71
记录语句，61–66
存储例程
定义，61，66
DEFINER 从句，66
记录语句，61–66
对象定义，180
返回值，69
字符串数据，解释，94
stunnel 命令
功能，203
复制支持，204–206
Sun Management Center，252
Sun Microsystems，437
SUPER 权限
配置复制，16
禁用，99
记录语句，65
设置线程 ID，57
存储函数，70
交换技术，249
sync-binlog 选项，83，100，390
同步
I/O 线程，224
中继服务器，188
slave 服务器，128，150，222–225
SQL 线程，224
故障排除，396
异步复制
异步复制，150
性能问题，151
sync_with_master 函数，186，192
SYSDATE 函数，53
System Health Report，277，278–280，285
System Profiler，268–271

T

表 ID，236
表，56
（参见临时表）
AUTO_INCREMENT 列，52，53，123
压缩，345，347
数据分片，170，176
碎片整理，348
非事务性改变，72–75，78
再植，564
安全问题，64
按索引顺序存储，347
故障排除，391，395，397
性能调整，345–346
表空间
定义，353
监控，363
Table_map 事件，232，234，236
tar 工具，428
Task Manager，285

Task Scheduler，42
临时表
　非事务性改变，79
　进程 ID，56
　pseudothread ID，221
　线程 ID，56
　故障排除，396
TEMPTABLE 视图，122
Terremark（供应商），487
抖动，249
线程 ID
　功能，56
　记录查询，52
　TLS 支持，220
线程局部存储（TLS），220
线程
　复制，200，371–372
　安全问题，70
　半同步复制，116
　slave，201，210–212
　事务缓存，77
3Tera（供应商），486
时间戳
　记录语句，52，53，61
　mysqlbinlog 支持，90，93
TLS（线程局部存储），220
top 命令，253，254–255
拓扑
　最佳实践，401–403
　检查服务器状态，403
　环形复制，142–146，152
　定义，23
　双主设置，6，23，115–124，152
　热备份，11，111–114
　管理，152–159
　移除 slave，109
　树，23，152
tps（每秒事务数），262
事务缓存，76–79
事务协调器，537
事务管理器，79
事务计算，482
事务，225
　（参见非事务性改变）
　异步复制，150
　隐式提交，135
　记录日志，75–81
　MySQL 集群，537
　半同步复制，124
　序列化执行，46
　Slave 服务器，222–225
　存储过程，141
　故障排除，397
　两阶段提交，150
每秒事务数（tps），262
树形拓扑
　描述，152
　管理复制，23
触发器
　创建，63
　DEFINER 从句，97
　事件，238–239
　调用，65
　记录语句，61–66
故障排除
　最佳实践，401–407
　二进制日志，392
　二进制日志事件，389–391
　数据丢失，396
　master 服务器，388–393

内存，395
非事务性改变，397
查询，391，392，394
Query Analyzer，463
中继日志，397
复制，393，398，399，406
slave 服务器，393–398，563
异步，396
表，391，395，397
临时表，396
事务，397
Tuckfield，Paul，574
两阶段提交，150

U

UAC（用户账户控制），42，277
UDFs（用户自定义函数），218，229
UML（统一建模语言），173
umount 命令，435
下划线（_），164
统一建模语言（UML），173
Unix 环境
自动监控，268
磁盘使用情况，261–264
一般系统统计数据，266
InnoDB Hot Backup 应用，425
管理复制，24
内存使用情况，259–261
监控，246，253–268
网络活动，265
进程活动，253–258
调度任务，42
UNIX_TIMESTAMP 函数，52，53
UNLOCK TABLES 命令，436，514
UPDATE 语句
LIMIT 从句，229，240
记录日志，50
非事务性改变，226
存储过程，66
内存表的故障排除，395
使用示例，134
WHERE 从句，46，50
Update_rows 事件，232，237
uptime 命令，252，253，266
USE 语句
当前数据库，60
使用示例，90
用户账户控制（UAC），42，277
User 类，25
USER 函数，229，231
用户自定义函数（UDFs），218，229
User_var 事件
功能，54，218
mysqlbinlog 支持，90，91
UUID 函数，231

V

Vagabond 角色，29
变量，306
（参见特殊变量类型）
二进制日志，98–100
配置服务器，293
非事务性改变，79
口令问题，63，64
查询事件，51，218
线程特定的结果，220
variables 命令，301
--verbose 选项，564
验证过程，417
vgcreate 命令，434

vgscan 命令，434
视图
 最佳实践，336
 优化，122
虚拟化，云计算，481
vmstat 命令，253，264，267
卷组，434
卷影复制，432

W

wait_for_pos 函数，186，191
wait_for_trans_id 函数，190
WHERE 从句
 DELETE 语句，46
 EXPLAIN 命令，322
 SELECT 语句，336
 UPDATE 语句，46，50
通配符
 mysqlbinlog 支持，92
 slave 过滤器，164
Windows 环境
 Cygwin，429
 Event Viewer，281–283
 InnoDB Hot Backup 应用，425
 管理复制，24
 监控，246，276–288
 Performance Monitor，285–288
 Reliability Monitor，283
 System Health Report，277，278–280，285
 Task Manager，285
 虚拟化，481
 卷影复制，432
 Windows Experience 报告，277
Windows Experience 报告，277
Windows Vista 环境
 监控，276–288
 调度任务，42
最坏的情况，413，416
Write_rows 事件，232，237
写数据
 数据分片，149
 负载均衡，148
 向外扩展，149
 线程局部变量，220

X

X.509 证书，498
X/Open Distributed Transaction Processing 模式 XA，79–81
XA 协议，79–81，83
Xid 事件，81，91
XtraBackup，432，437
XtraDB，432

Z

Zawodny，Jeremy D.，316
ZFS（Zettabyte File System），34，437

关于作者

Charles Bell 博士，Oracle 高级软件工程师。现任备份开发组组长，同时是 MySQL Backup and Replication 小组的成员。他与自己心爱的妻子住在弗吉尼亚乡村的一个小镇上。2005 年获得弗吉尼亚联邦大学工程学博士学位。研究兴趣包括数据库系统，版本系统，语义 web，及敏捷软件开发。

Mats Kindahl 博士，MySQL server 方面的高级软件开发人员，是 MySQL 基于行的复制的主要架构师和实现者，并负责复制的战略开发，以及再工程和 plug-in 架构。在 MySQL 工作之前，他主要从事正规化方法、程序分析和分布式系统等方面的研究，并获得计算机科学博士学位。具有多年的 C/C++ 编译器的开发经验，熟悉多种编程语言。

Lars Thalmann 博士，MySQL 复制与备份的开发经理，负责这些特性的战略和开发，带领整个工程师团队。2001 年 Thalmann 是 MySQL 集群的软件开发人员，此后一直从事 MySQL 开发。近年来，他创建并开发了 MySQL Enterprise Backup 特性，自 2004 年就带领 MySQL 复制团队，已经成为 MySQL 集群复制开发的关键人物。Thalmann 获得瑞典乌普萨拉大学计算机科学博士学位。

出版商商标

“高可用 MySQL——构建健壮的数据中心”封面上的动物是美洲知更鸟（美洲鸫）。其相貌独特，很容易辨认——黑色的头部，橘红色的胸部，褐色的背部——它属于画眉鸟类，是最常见的美洲鸟类之一。（虽然它与欧洲知更鸟名字一样，而且欧洲知更鸟的胸部也是红色的，但是这两个物种并没有多大关系。）

美洲知更鸟栖居于北美六百万平米的范围，美国基本上全年都有这种鸟。知更鸟通常被认为是春季的先兆，它们一大早就开始唱歌，一直到别的鸟儿都睡了还会鸣叫。它们吃无脊椎动物（通常是蚯蚓）和水果、浆果类。知更鸟喜欢开阔的地方和不高的草地，所以它们常常飞去后院，也常常出现在公园、花园和草地等场所。

封面上的图片来自“Johnson’s Natural History”第二卷。

关于译者

宁青

工作过几年，由一家小公司转战 Cisco WebEx，现在淘宝数据平台部架构组下讨生活。做过几年 Java，并会点 Perl，Python。现专注于数据这块，搞过 Oracle 和 MySQL，顺带还考了个 OCP 认证，但绝大多数时间是在 Hive，Hadoop 这块瞎搞。

唐李洋

合肥工业大学在读博士，研究方向为云计算、并行计算、数据挖掘与分析等。曾经在 Cisco WebEx 实习，主要从事数据架构方面的设计与分析工作。

诸云萍

合肥工业大学计算机专业学士，中国科学技术大学工商管理硕士（MBA），Oracle 认证 OCP。从事数据库开发、测试与管理工作 5 年以上，具有丰富的 MySQL，Oracle，Sybase 等关系型数据库项目实践经验。曾在神州数码安徽移动 BOSS 项目组负责报表开发工作，现就职于 Cisco-WebEx 合肥分公司，从事 Oracle，Mysql 数据库以及 ETL 的测试与管理工作。